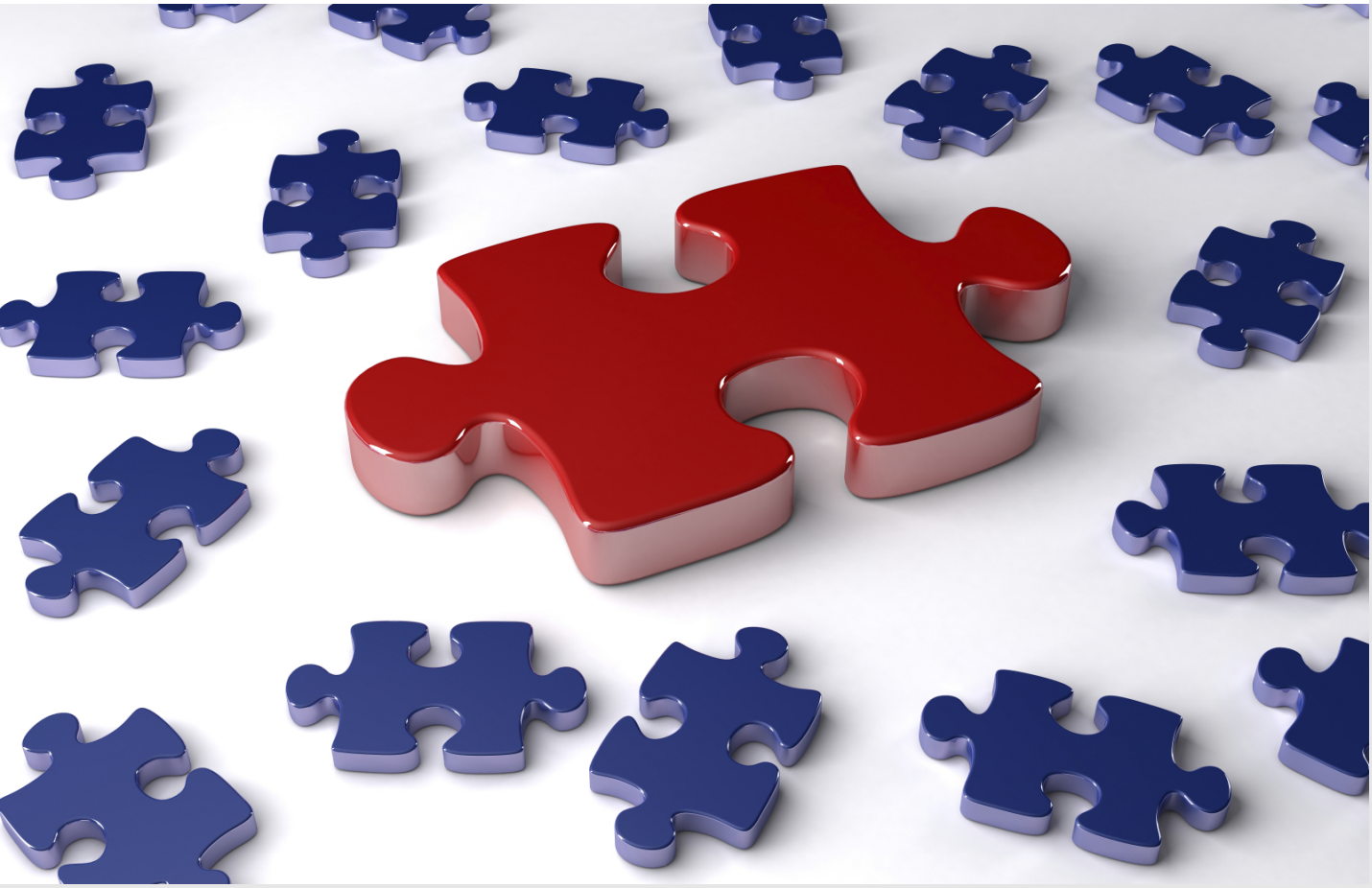
TU Clausthal



Holger Klus

Anwendungsarchitektur-konforme Konfiguration selbstorganisierender Softwaresysteme

SSE-Dissertation 8

Software
Systems
Engineering

Institut für Informatik
Lehrstuhl von Prof. Dr. Andreas Rausch

Anwendungsarchitektur-konforme Konfiguration selbstorganisierender Softwaresysteme

Dissertation

zur Erlangung des Grades eines Doktors
der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von

Holger Klus
aus Nordhorn

Genehmigt von der Fakultät für Mathematik/Informatik und Maschinenbau der
Technischen Universität Clausthal

2013

Vorsitzender der Prüfungskommission
Prof. Dr. Niels Pinkwart

Hauptberichterstatter
Prof. Dr. Andreas Rausch

Berichterstatter
Prof. Dr. Jörg P. Müller

Tag der mündlichen Prüfung: 15.03.2013

SSE-Dissertation 8, 2013

Kurzbeschreibung

Rechner sind in den vergangenen Jahren und Jahrzehnten immer kleiner und gleichzeitig immer leistungsfähiger geworden. Im Zuge dieser Entwicklung hat die Verbreitung verteilter mobiler Anwendungen stark zugenommen. Hierbei handelt es sich um Anwendungen, die aus zahlreichen, miteinander kommunizierenden Komponenten bestehen, welche auf unterschiedlichen, teils mobilen Rechnern wie Smartphones, Tablet-PCs oder Laptops ausgeführt werden.

Bei der Entwicklung derartiger Anwendungen müssen besondere Anforderungen berücksichtigt werden. Unter anderem müssen die Anwendungen in der Lage sein, auf den Ausfall einzelner Komponenten reagieren zu können, beispielweise verursacht durch einen leeren Akku. Auch können Verbindungen zwischen Komponenten jederzeit unterbrochen werden, zum Beispiel verursacht durch ein Funkloch. Umgekehrt können neue Komponenten im Netzwerk verfügbar werden, welche in eine laufende Anwendung integriert werden müssen. Die Anwendung muss hierbei in der Lage sein, selbstständig und zur Laufzeit auf derartige Ereignisse zu reagieren, um den Endanwendern eine möglichst unterbrechungsfreie Nutzung zu ermöglichen.

Die Entwicklung verteilter mobiler Anwendungen ist auf Grund dieser Anforderungen aufwändig und komplex. Deshalb werden bereits seit einigen Jahren Möglichkeiten erforscht, die Entwicklung derartiger Anwendungen zu erleichtern. Ein vielversprechender Ansatz ist hierbei die Verwendung von Konzepten selbstorganisierender Softwaresysteme.

In der vorliegenden Arbeit wird eine Lösung vorgestellt, welche die Entwicklung und Ausführung mobiler verteilter Anwendungen basierend auf selbstorganisierenden Komponenten ermöglicht. Hierbei kommt der Ansatz ohne zentrale Konfigurationseinheit aus und ermöglicht zudem ein emergentes Systemverhalten.

Anwendungen, welche auf Basis selbstorganisierender Komponenten entwickelt werden, besitzen den Nachteil, dass sie in erster Linie komponentenlokale Anforderungen berücksichtigen. So kann ein System entstehen, in dem die Anforderungen aller Komponenten erfüllt sind, das System selbst aber keinen Nutzen erbringt, da anwendungsspezifische Anforderungen unberücksichtigt bleiben.

In dieser Arbeit wird deshalb die Lösung zur Entwicklung und Ausführung verteilter mobiler Anwendungen auf Basis selbstorganisierender Komponenten erweitert um die Möglichkeit, anwendungsspezifische, komponentenübergreifende Anforderungen spezifizieren zu können. Eine Besonderheit der Lösung ist die Spezifikation von komponentenübergreifenden Anforderungen in Form einer Anwendungsarchitektur ohne Kenntnis der konkreten Komponenten. Auf diese Weise kann der emergente Charakter selbstorganisierender Systeme erhalten werden, bei gleichzeitiger Berücksichtigung Anwendungsarchitektur-spezifischer, komponentenübergreifender Anforderungen. Abschließend wird ein Framework vorgestellt, welches die genannten Konzepte prototypisch realisiert. Der Lösungsansatz wird zudem anhand eines typischen verteilten mobilen Softwaresystems evaluiert, welches zum Ziel hat, das Training von Biathleten zu unterstützen.

Danksagung

Auf dem langen Weg zur Fertigstellung dieser Arbeit haben mich viele Menschen begleitet und unterstützt. Bei denen möchte ich mich an dieser Stelle sehr herzlich bedanken.

Zunächst möchte ich mich bei Prof. Dr. Andreas Rausch bedanken, der mich während der vergangenen Jahre stets unterstützt und ermutigt hat, das Thema der Anwendungsarchitektur-Konformität in selbstorganisierenden Systemen anzugehen und hierzu eine Lösung zu erarbeiten. Die zahlreichen Diskussionen mit ihm zu diesem Thema habe ich als sehr hilfreich empfunden, und haben immer wieder zu neuen Denkanstößen geführt.

Des Weiteren bedanke ich mich bei Prof. Dr. Jörg P. Müller für seine Bereitschaft als Zweitgutachter zu fungieren, sowie für seine hilfreichen Kommentare und Verbesserungsvorschläge.

Zahlreiche Kollegen haben mich in den letzten Jahren auf dem Weg zu dieser Arbeit begleitet. Allen voran möchte ich mich auf diesem Wege bei den beiden Kollegen bedanken, mit denen ich im Laufe der Jahre ein Büro geteilt habe, nämlich Dirk Niebuhr und Sebastian Herold. Auch in stressigen Phasen waren sie stets bereit, verschiedene Fragestellungen meiner Arbeit mit mir zu diskutieren. Des Weiteren möchte ich Dirk und Sabine Niebuhr sowie Benjamin Fischer für das Korrekturlesen dieser Arbeit danken.

Mein besonderer Dank gilt meinen Eltern, die mich während meiner akademischen Ausbildung stets unterstützt und ermutigt haben. Außerdem möchte ich mich bei meinen Freunden bedanken, die immer wieder für freizeitliche Ablenkung gesorgt haben, sei es durch gemeinsame sportliche Aktivitäten, gemeinsame abendliche Kneipengänge, Spiele- und Filmabende und nicht zuletzt durch die Gründung unserer kleinen Lehrstuhlband.

Ohne die Unterstützung von euch allen hätte ich diese Arbeit nicht schreiben können. Deshalb nochmals vielen Dank für alles!

Holger Klus

Inhaltsverzeichnis

1 Einleitung

In den vergangenen Jahren und Jahrzehnten gab es einen rasanten technologischen Fortschritt in der Computerindustrie. So hat sich die Rechenleistung seit den 60er Jahren alle 18 bis 24 Monate verdoppelt, so wie es Gordon Moore bereits 1965 in Form des Moore'schen Gesetzes prognostiziert hatte [Moo65]. Gleichzeitig sind die Rechner immer kleiner, leichter und mobiler geworden. Hinzu kamen stetig wachsende Möglichkeiten, diese Rechner untereinander zu vernetzen. Die Anwendungsmöglichkeiten, die sich durch diesen technologischen Fortschritt für die Entwicklung von Anwendungen ergeben, sind sehr vielfältig und haben sich im Laufe der Jahre und Jahrzehnte ständig verändert und weiterentwickelt.

Insbesondere in den vergangenen Jahren haben sich durch die Möglichkeiten der Vernetzung kleiner und gleichzeitig leistungsstarker Rechner immer neue Möglichkeiten aufgetan, die Menschen im Alltag zu unterstützen. Durch die Verbreitung des Internets sind Menschen beispielsweise in der Lage, jederzeit vom Wohnzimmer aus einzukaufen, Banküberweisungen zu tätigen, mit Freunden zu chatten, Emails auszutauschen und vieles mehr. Mit der Entwicklung mobiler Plattformen wie Laptops, Tablet-PCs und internetfähiger Smartphones, zusammen mit einer immer besseren Netzabdeckung durch Mobilfunkanbieter, steht einer Vernetzung überall und jederzeit nichts mehr im Wege. So können nun während eines Ausflugs mit Hilfe des Smartphones die nächstgelegenen Restaurants gefunden, deren Speisekarten eingesehen, und gegebenenfalls ein Tisch reserviert werden. Während eines Stadtrundgangs können vor Ort Informationen über die Sehenswürdigkeiten eingesehen werden und Öffnungszeiten von Museen abgefragt werden.

Neben den Möglichkeiten, die die Mobilität heutiger Rechner bietet, können auch kleine Sensor- und Aktuatoreinheiten mit Kommunikationsfunktionalität die Möglichkeiten von Softwaresystemen erweitern. So besitzen einige Smartphones einen Näherungssensor, welcher erkennt, ob ein Nutzer das Telefon ans Ohr hält und können z.B. durch Deaktivierung aller für das Telefonieren nicht relevanter Funktionen reagieren. GPS-Sensoren können zur Realisierung von Navigationssoftware verwendet werden, aber auch zur ortsabhängigen Bereitstellung von Diensten. So könnte eine Restaurant-App lediglich diejenigen Restaurants anzeigen, welche sich in unmittelbarer Umgebung zum aktuellen Aufenthaltsort befinden. Die sogenannten kontextsensitiven Systeme gehen hier noch einen Schritt weiter. Hier werden nicht nur Ortsinformationen, sondern sämtliche, über die aktuelle Ausführungsumgebung bekannten Informationen durch die Anwendung verwendet, um diese der Umgebung anzupassen. So könnte eine kontextsensitive Telefonanwendung beispielsweise automatisch erkennen, wo sich der Anwender gerade befindet. Erkennt die Anwendung, dass sich der Nutzer gerade in einem Meeting befindet, könnte sie automatisch auf einen Vibrationsalarm umschalten. Kontextinformationen werden hierbei durch unterschiedlichste Sensoren und Softwaredienste bereitgestellt.

1.1 Motivation

Insbesondere durch die Verfügbarkeit immer kleiner und gleichzeitig günstiger werdender Sensorik sind weitere Anwendungsgebiete entstanden. Hierzu gehört u.a. das Konzept der Ambient Intelligence. Ziel dieser Systeme ist es, Sensoren, Aktuatoren und Rechner derart zu vernetzen, dass der Alltag verbessert werden kann. Eines der Anwendungsgebiete ist das intelligente Haus, bei dem sämtliche Einrichtungen wie Rollläden, Beleuchtung, Kühlschrank

usw. über mobile Computer wie Smartphones gesteuert werden können. Ein weiteres Anwendungsgebiet ist das Assisted Living, bei dem ein Softwaresystem ältere und auch benachteiligte Menschen situationsabhängig und unaufdringlich unterstützt. Dies kann z.B. darin bestehen, ältere Menschen automatisch an das Trinken zu erinnern, oder automatisch eine Einsatzzentrale zu benachrichtigen, für den Fall, dass ein Bewohner gestürzt ist.

Anwendungen, wie sie soeben vorgestellt wurden, bestehen aus einer Vielzahl unterschiedlicher, miteinander interagierender Komponenten, seien es reine Hardwarekomponenten zur Datenerfassung wie Sensoren, oder Komponenten zur Verarbeitung von Daten, zur Entgegennahme von Benutzereingaben oder zur Darstellung von Informationen. Eine besondere Herausforderung hierbei ist, diese Komponenten miteinander zu vernetzen, so dass ein Informationsaustausch möglich wird. So benötigt beispielsweise eine grafische Benutzungsschnittstelle zur Anzeige der aktuellen Raumtemperatur Zugriff auf einen Sensor, der die aktuelle Raumtemperatur misst und über eine Schnittstelle zur Verfügung stellt. Die Komponente könnte zudem Zugriff auf das Thermostat der Heizung benötigen, um dem Nutzer die Möglichkeit zu geben, die Einstellungen anzupassen.

Eine besondere Schwierigkeit bei der Entwicklung dieser Anwendungen ist, dass einzelne Komponenten oder auch Kommunikationsverbindungen zwischen Komponenten jederzeit ausfallen können. So kann es beispielsweise vorkommen, dass das GPS-Signal einer Ortungskomponente zu schwach geworden ist, um die aktuelle Position zu bestimmen. Dies kann wiederum dazu führen, dass Komponenten, welche diese Ortungskomponente nutzen, ggf. nicht mehr korrekt funktionieren. Hier könnte z.B. dann dadurch Abhilfe geschaffen werden, dass von GPS-basierter Ortung auf WLAN-basierte Ortung gewechselt wird. Ebenso müsste eine erneute Anpassung vorgenommen werden, wenn das GPS-Signal wieder zur Verfügung steht.

Allerdings ist es für die zuvor beschriebenen Anwendungen nicht praktikabel, derartige Anpassungen manuell durchführen zu müssen: Gerade in mobilen Einsatzszenarios können derartige Ausfälle häufig auftreten. Der Anwender wäre einen Großteil der Zeit damit beschäftig, die Anwendung derart zu konfigurieren, dass sie wunschgemäß funktioniert. Hier müssen Lösungen gefunden werden, die derartige Anpassungen automatisiert durchführen. Vielversprechende Lösungsansätze liefern hierbei Konzepte aus dem Bereich der selbstorganisierenden Systeme. Diese berücksichtigen insbesondere die selbständige Reaktion auf äußere Einflüsse, wie z.B. den Ausfall einer oder mehrere Komponenten. Hierzu beobachten diese Systeme selbständig die Umgebung, analysieren die erfassten Informationen und Veränderungen in der Umgebung, planen geeignete Lösungsstrategien und setzen sie autonom um.

Für die zuvor beschriebenen verteilten, mobilen Anwendungen besteht eine Form der Umsetzung darin, jede Komponente als selbstorganisierende Einheit zu realisieren. Diese ist dann in der Lage, autonom z.B. auf den Wegfall oder das Hinzukommen von Komponenten zu reagieren, von denen sie abhängt. Stehen einer Komponente nicht mehr alle benötigten Komponenten zur Auflösung der Abhängigkeiten zur Verfügung, so stoppt sie die Ausführung. Darauf müssen ggf. wiederum andere Komponenten autonom reagieren. Stehen einer Komponente beispielsweise mehr Komponenten zur Verfügung als benötigt werden, so kann sie diejenigen zur Verwendung auswählen, die aus Sicht der Komponente als am geeignetsten eingestuft werden. Auf diese Weise entsteht automatisch ein System basierend auf interagierenden, selbstorganisierenden Komponenten.

Ein System, welches aus selbstorganisierenden Komponenten aufgebaut wird, besitzt zahlreiche Vorteile. So kann z.B. auf eine schwergewichtige zentrale Infrastruktur verzichtet werden, die sich um die Konfiguration des Systems kümmert. Die Realisierung einer solchen zentralen Konfigurationseinheit ist zum einen sehr fehleranfällig aufgrund der meist hohen Komplexität des Konfigurationsalgorithmus. Außerdem entsteht ein hoher Netzwerkverkehr zwischen dieser zentralen Einheit und den einzelnen zu konfigurierenden Komponenten. Und schließlich wirken sich Fehler oder der Ausfall der zentralen Einheit auf die Funktionsfähigkeit des gesamten Systems aus. Diese Gründe sprechen dafür, verteilte mobile Anwendungen auf Basis eines dezentralen Konfigurationsmechanismus zu konstruieren.

Allerdings birgt der Ansatz auch einige Nachteile. Denn das System entwickelt sich auf Basis lokaler Konfigurationsentscheidungen einzelner, selbstorganisierender Komponenten. Da zur Entwicklungszeit der Komponenten häufig nicht bekannt ist, für welchen Einsatzzweck die Komponenten verwendet werden, basieren die Entscheidungen meist darauf, stets eine Entscheidung zu treffen, die für die Komponente die beste Lösung darstellt. Dies betrifft z.B. die Auswahl der Komponenten, die eine Komponente zur Ausführung seiner Dienste benötigt. Allerdings kann dieser Ansatz zu Systemen führen, bei denen zwar lokal die jeweils beste Konfigurationsentscheidung getroffen wurde, das System als Ganzes jedoch den Anforderungen nicht gerecht wird. Ein weiterer Aspekt betrifft die Auswahl von Komponenten, die für eine Anwendung benötigt werden. Angenommen, eine Anwendung soll aus zwei Komponenten bestehen, welche so konstruiert sind, dass sie auch unabhängig voneinander ausgeführt werden können. Wird nun zunächst eine der Komponenten im System installiert, so wird sie alle Abhängigkeiten zu anderen Komponenten auflösen und mit ihrer Ausführung beginnen, unabhängig davon, ob die zweite Komponente bereits zur Verfügung steht. Es fehlt in dieser Art von Systemen die Möglichkeit, auf den Konfigurationsprozess des Systems als Ganzes derart einzuwirken, dass Anwendungsarchitektur-spezifische Anforderungen berücksichtigt werden. Dies sind Anforderungen, die über die Anforderungen einzelner Komponenten hinausgehen und üblicherweise vom Anwendungsentwickler definiert werden.

Wünschenswert wären somit mobile verteilte Anwendungen, welche auf der einen Seite die Vorteile selbstorganisierender Komponenten nutzen, und zum anderen die Möglichkeit bieten, anwendungsspezifische, komponentenübergreifende Anforderungen automatisiert umzusetzen. Hierbei sollte möglichst auf eine schwergewichtige zentrale Konfigurationseinheit verzichtet werden. Auf diese Weise ließen sich Anwendungen erzeugen, die aus unterschiedlichsten Komponenten zusammengesetzt sind, von unterschiedlichen Entwicklern entwickelt wurden, auf verschiedenen Geräten in einem Netzwerk verteilt ausgeführt werden, selbstständig auf den Wegfall oder das Hinzukommen von Komponenten reagieren, und dabei anwendungs-spezifische Anforderungen berücksichtigen. Ziel dieser Arbeit war es, hierfür eine Lösung zu erarbeiten. Die einzelnen Teilziele werden nun im folgenden Abschnitt genauer beschrieben.

1.2 Ziele und Beitrag

Die Entwicklung mobiler verteilter Anwendungen ist eine komplexe Aufgabe. Dies liegt insbesondere an den Anforderungen, die an derartige Anwendungen gestellt werden. In dieser Arbeit wird eine Lösung vorgestellt, welche den Entwicklern die Erstellung und Ausführung von Anwendungen unter Berücksichtigung dieser besonderen Anforderungen ermöglicht. Auf Grund der Vielfältigkeit möglicher Anforderungen wurden in dieser Arbeit folgende ausgewählte Aspekte betrachtet:

- Automatisierte Konfiguration zur Laufzeit
- Berücksichtigung komponentenlokaler Anforderungen
- Berücksichtigung Anwendungsarchitektur-spezifischer Anforderungen

Diese Aspekte werden nun im Folgenden kurz erläutert.

1.2.1 Automatisierte Konfiguration zur Laufzeit (A1)

Gerade mobile verteilte Anwendungen sind durch einen häufigen Wechsel der Ausführungsumgebung geprägt, was ggf. zu einem häufigen Bedarf an Systemanpassungen führt. Hierbei ist es nicht praktikabel, derartige Anpassungen durch den Nutzer oder einen Administrator durchführen zu lassen.

Ziel ist es deshalb, eine Infrastruktur bereitzustellen, welche auf bestimmte Ereignisse automatisch und zur Laufzeit reagieren kann. Die in dieser Arbeit betrachteten Ereignisse sind zum einen das Hinzukommen und der Wegfall von Komponenten, und zum anderen die Veränderung des Zustandes von Komponenten.

Die Umsetzung einer automatisierten Konfiguration zur Laufzeit ist in den meisten Fällen sehr komplex, was dazu führt, dass häufig ein Großteil der Entwicklungszeit darauf verwendet wird, die Konfigurationslogik zu realisieren, obwohl eigentlich die Implementierung fachlicher Anforderungen im Vordergrund stehen sollte. Aus diesem Grunde soll im Rahmen dieser Arbeit ein Framework bereitgestellt werden, welches Aufgaben der Selbstorganisation weitestgehend übernimmt und hierbei gleichzeitig anwendungs- und komponentenspezifische Anforderungen berücksichtigt.

1.2.2 Berücksichtigung komponentenlokaler Anforderungen (A2)

In dieser Arbeit werden Systeme betrachtet, die ein emergentes Verhalten aufweisen. Emergente Software kombiniert dynamisch und flexibel die vorhandenen Komponenten unterschiedlicher Hersteller, um den komplexen Anforderungen gerade in mobilen verteilten Anwendungen gerecht zu werden. Ein solches System wird nicht als Ganzes entwickelt, ausgeliefert und ausgeführt, sondern entsteht erst aus dem Zusammenwirken seiner unabhängig bereitgestellten Teile.

Ziel dieser Arbeit ist es, ein Konzept für die Realisierung emergenter Software zu entwickeln. Dieses basiert darauf, ein System mit Hilfe von selbstorganisierenden Komponenten aufzubauen. Diese Komponenten müssen in der Lage sein, sich selbst ohne die Mitwirkung einer zentralen Konfigurationseinheit zur Laufzeit zu managen. Eine der wesentlichen Konfigurationsaufgaben einer Komponente besteht darin, Abhängigkeiten zu anderen Komponenten aufzulösen, und hierbei komponentenspezifische Anforderungen zu berücksichtigen. Hierbei müssen selbstorganisierende Komponenten zunächst die Fähigkeit besitzen, auf den Wegfall verwendeter Komponenten zu reagieren, sowie ggf. neue Komponenten nutzen zu können, ohne dass hierzu ein manueller Eingriff in das System notwendig wird. Diese Konfigurationsfunktionalität soll durch ein Framework weitestgehend übernommen werden. Hierbei soll jedoch dem Komponentenentwickler die Möglichkeit gegeben werden, komponentenspezifisch Einfluss auf das Konfigurationsverhalten zu nehmen. So soll der Komponentenentwickler die Möglichkeit besitzen, Anforderungen an benötigte Komponenten festlegen zu können. Zu diesen Anforderungen gehören unter anderem benötigte Schnittstellen,

Anforderungen an den Zustand der zu verwendenden Komponenten, sowie die minimale und maximale Anzahl benötigter Komponenten.

1.2.3 Berücksichtigung Anwendungsarchitektur-spezifischer Anforderungen (A3)

Wie bereits erwähnt wurde, besteht ein Ziel der Arbeit darin, möglichst auf eine zentrale Konfigurationseinheit zu verzichten. Hierdurch können komponentenübergreifende Anforderungen allerdings nicht ohne Weiteres berücksichtigt werden. Dies ist in vielen Fällen jedoch nicht akzeptabel.

Ziel ist es deshalb, den dezentralen Aufbau der Infrastruktur möglichst beizubehalten, und gleichzeitig Einfluss auf den Konfigurationsprozess nehmen zu können, so dass anwendungsspezifische Anforderungen erfüllt werden. Der Fokus dieser Arbeit liegt hierbei darin, Anforderungen hinsichtlich der Anwendungsarchitektur zu berücksichtigen. Hierbei muss dem Anwendungsentwickler die Möglichkeit gegeben werden, Anforderungen bzgl. der Auswahl von Komponenten für die Anwendung sowie bzgl. der Verbindungen zwischen diesen definieren zu können. Eine weitere Herausforderung hierbei ist es, das emergente Verhalten möglichst zu erhalten, und somit auch die Integration von Komponenten, die zur Entwicklungszeit der Anwendung noch nicht bekannt waren, zu ermöglichen.

Die Problemstellung wird in Kapitel 3 anhand eines kleinen Anwendungsbeispiels nochmals detailliert erläutert. In den folgenden Kapiteln wird die hierzu erarbeitete Lösung vorgestellt. Hierzu werden zunächst die Lösungskonzepte technologieneutral erläutert, und anschließend eine mögliche Realisierung in Form eines Implementierungskonzeptes dargestellt. Die Gliederung der Arbeit wird im folgenden Abschnitt noch einmal im Detail vorgestellt.

1.3 Inhalt und Gliederung

Die Arbeit gliedert sich in neun Kapitel, welche in Abbildung 1-1 im Überblick dargestellt sind.

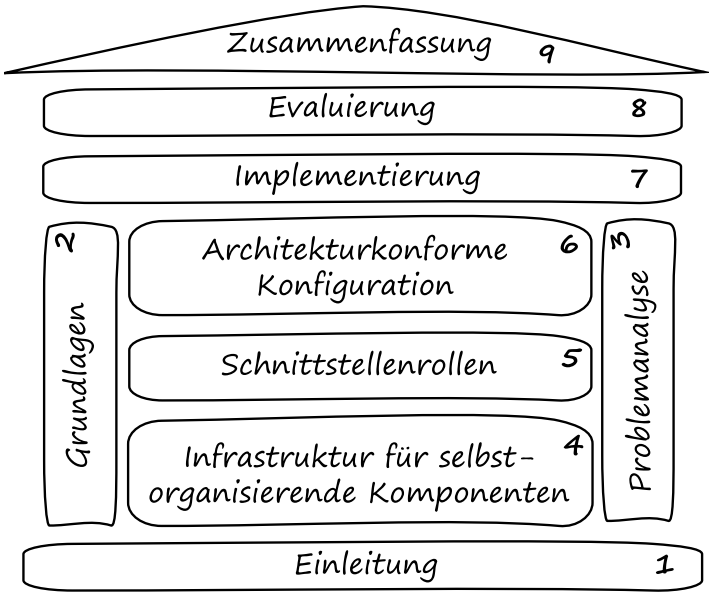


Abbildung 1-1: Überblick über die Kapitel der Arbeit

Im folgenden Kapitel werden zunächst die Grundlagen vermittelt, welche zum Verständnis der Problemstellung sowie der hier vorgestellten Lösung notwendig sind. Insbesondere werden hier die Grundlagen selbstorganisierender Softwaresysteme beschrieben, deren Anwendungsgebiete sowie existierende Frameworks.

In Kapitel 3 erfolgt dann basierend auf den Grundlagen eine detaillierte Problemanalyse anhand eines einfachen Anwendungsbeispiels, nämlich einer Anwendung zur Trainingsunterstützung für Biathleten. Anhand dieses Beispiels werden zum Abschluss des Kapitels einige verwandte Arbeiten vorgestellt, die sich ebenfalls mit dem hier behandelten Problem auseinandersetzen.

In den Kapiteln 4, 5 und 6 wird schrittweise die erarbeitete Lösung erläutert. Hierbei wird in Kapitel 4 zunächst eine Infrastruktur zur Entwicklung und Ausführung selbstorganisierender Systeme vorgestellt. Die in diesem Kapitel beschriebene Lösung eignet sich, um ein selbstorganisierendes System basierend auf selbstkonfigurierenden Komponenten zu entwickeln und auszuführen. Hierbei werden allerdings komponentenübergreifende, anwendungsspezifische Anforderungen zunächst nicht berücksichtigt.

In Kapitel 5 wird die in Kapitel 4 vorgestellte Infrastruktur dahingehend erweitert, dass bei der Auflösung von Abhängigkeiten zu anderen Komponenten auch der Komponentenzustand in den Konfigurationsprozess einfließt. Die beschriebene Erweiterung bietet so z.B. die Möglichkeit sicherzustellen, dass eine Restaurant-Such-Komponente nur mit denjenigen Restaurantkomponenten verbunden wird, welche ein geöffnetes Restaurant repräsentieren.

In Kapitel 6 wird wiederum aufbauend auf der Lösung in Kapitel 5 eine Erweiterung vorgestellt, die die Spezifikation und Berücksichtigung komponentenübergreifender, anwendungsspezifischer Anforderungen ermöglicht.

In Kapitel 7 wird die prototypische Implementierung der Infrastruktur im Überblick vorgestellt. Hierbei handelt es sich um eine Java-basierte Implementierung, welche sowohl die Konzepte aus Kapitel 4 und 5 realisiert, als auch ein Implementierungskonzept der in Kapitel 6 vorgestellten Lösung bereitstellt.

In Kapitel 8 erfolgt abschließend eine szenariobasierte Evaluierung der erarbeiteten Lösungskonzepte. Hierbei wurde für jede der in Kapitel 4, 5 und 6 vorgestellten Erweiterung eine eigene Evaluierung vorgenommen.

In Kapitel 9 folgen abschließend die Zusammenfassung der vorgestellten Ergebnisse sowie ein kurzer Ausblick.

2 Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die zur Verdeutlichung der Problemstellung sowie zum Verständnis der erarbeiteten Lösung notwendig sind. Hauptgegenstand dieser Arbeit sind selbstorganisierende Softwaresysteme. Die Grundlagen hierzu werden in Abschnitt 2.2 erläutert. Im darauffolgenden Abschnitt wird eine verbreitete Referenzarchitektur für selbstorganisierende Softwaresysteme vorgestellt, an welcher sich auch der erarbeitete Lösungsansatz orientiert. In Abschnitt 2.4 werden abschließend einige Frameworks zur Realisierung selbstorganisierender Softwaresysteme zusammengefasst.

Selbstorganisierende Softwaresysteme sind meist auch komponentenbasierte Anwendungen. Hierbei handelt es sich um Anwendungen, die aus mehreren unterschiedlichen Komponenten zusammengesetzt sind, welche zur Laufzeit miteinander interagieren, um die Anwendungsfunktionalität zu realisieren. Die für diese Arbeit wesentlichen Grundlagen komponentenbasierter Softwaresysteme werden im nun folgenden Abschnitt vorgestellt.

2.1 Komponentenbasierte Softwareentwicklung

Die Grundidee komponentenbasierter Softwareentwicklung ist es, ein komplexes System nicht als ein großes Modul zu entwickeln, sondern in Form von kleineren, miteinander interagierenden Komponenten. Die Vorteile der Modularisierung von Software wurden bereits sehr früh diskutiert und Kriterien zur Komponentenbildung vorgeschlagen [Par72], allerdings fehlte es lange an Möglichkeiten, diese sinnvoll umzusetzen. Im Folgenden wird zunächst erläutert, welche Eigenschaften eine Komponente im Allgemeinen besitzt.

2.1.1 Eigenschaften einer Komponente

Trotz der bereits langen Geschichte komponentenbasierter Anwendungsentwicklung existiert keine einheitliche Definition dessen, was eine Softwarekomponente genau ausmacht. In der Vergangenheit wurden bereits zahlreiche Versuche unternommen, diese Eigenschaften zusammenfassend zu formulieren [BDH+98]. Eine allgemein akzeptierte Definition beschreibt eine Komponente als Einheit zur Komposition mit folgenden Eigenschaften [Szy98, Szy02, HnP06]:

- Sie spezifiziert Schnittstellen für den Zugriff auf die realisierte Funktionalität
- Sie besitzt ausschließlich explizit definierte Abhängigkeiten zum Kontext
- Sie kann unabhängig von anderen Komponenten installiert werden
- Sie ist Gegenstand von Komposition durch Dritte

Zunächst realisiert eine Komponente in der Regel eine Menge von Funktionalitäten, die thematisch oder organisatorisch zusammengehören. Auf die internen Bestandteile einer Komponente, welche die angebotenen Funktionalitäten realisieren, gewährt eine Komponente keinen Zugriff. Vielmehr wird die Funktionalität über Schnittstellen nach außen bereitgestellt. Eine Schnittstelle definiert hierbei eine Menge von Operationen, zu denen weitere Informationen hinterlegt sind. Die Beschreibung einer Schnittstelle kann hierbei z.B. folgende Informationen beinhalten:

- Syntax (Name der Methoden, Parameter, Rückgabewerte)
- Semantik (Bedeutung der Methoden, realisierte Funktionalität der Methoden, Bedeutung der Rückgabewerte)
- Fehlerbehandlung (Fehlercodes und deren Bedeutung, Fehlerfälle)
- Konfigurierbarkeit (Anpassbarkeit des Verhaltens einer Schnittstelle)
- Qualitätseigenschaften (Mit welcher Qualität wird die Funktionalität realisiert: z.B. Antwortzeit, Genauigkeit, Zuverlässigkeit, …)
- Entwurfsentscheidungen (Begründung für die Zusammensetzung einer Schnittstelle)
- Version

Jede Komponente kann auf der einen Seite beliebig viele Schnittstellen anbieten, und auf der anderen Seite angebotene Schnittstellen anderer Komponenten nutzen. Der Anbieter einer Schnittstelle orientiert die Implementierung an den Vorgaben, die die Schnittstelle spezifiziert. Der Verwender einer Schnittstelle kann anhand der Schnittstellenspezifikation eine Auswahl eines geeigneten Anbieters treffen. Schnittstellen werden in der Regel unabhängig von Komponenten definiert [SoW98, ChD00]. Anbietende und nutzende Komponenten müssen sich zur Entwicklungszeit nicht kennen.

Die zweite zentrale Eigenschaft ist die, dass Komponenten explizit definierte Abhängigkeiten zur Umgebung besitzen. Derartige Abhängigkeiten werden in der Regel durch die Angabe von Schnittstellen definiert, für die die Komponente eine Implementierung benötigt, um ihrerseits ausgeführt werden zu können.

Die dritte Eigenschaft besagt, dass Komponenten unabhängig von anderen Komponenten installiert werden können. Komponenten sind somit eigenständige Einheiten, die in unterschiedlichen Anwendungen eingesetzt werden können. Bei deren Ausführung muss lediglich sichergestellt werden, dass die definierten expliziten Abhängigkeiten aufgelöst werden.

Komponenten werden idealerweise so entwickelt, dass sie in unterschiedlichen Anwendungen wiederverwendet werden können. Hierbei kann auch der Fall eintreten, dass eine Komponente unabhängig von einer spezifischen Anwendung entwickelt wurde. Ein Anwendungsentwickler hat hierbei die Aufgabe, die benötigten Komponenten ggf. einzukaufen und diese zur gewünschten Anwendung zu komponieren. Hierbei besteht die Hauptaufgabe darin, zu definieren, welche Komponente mit welchen anderen Komponenten verbunden werden soll. Der Vorgang wird auch als Konfiguration bezeichnet. Recht früh wurde ebenfalls der Ansatz verfolgt, die Beschreibung und Implementierung einzelner Komponenten einer Anwendung von der Konfiguration der Anwendung zu trennen. Dieses Prinzip ist unter dem Begriff *Configuration Programming* bekannt geworden [Kra90]. Hierbei werden im Wesentlichen vier Dinge gefordert. Zunächst sollten Konfigurationssprache und Komponenten-Programmiersprache getrennt sein. Zweitens sollten Komponenten als kontextunabhängige Typen mit wohldefinierten Schnittstellen definiert sein. Drittens sollten komplexe Komponenten definierbar sein als Komposition von Instanzen von Komponententypen. Und schließlich sollten Anpassungen auf Konfigurationsebene ausgedrückt werden in Form von Änderungen der Komponenteninstanzen sowie deren Verbindungen. CONIC [KrM85] war eines der ersten Systeme, bei dem dieses Prinzip umgesetzt wurde.

2.1.2 Grafische Notation

In dieser Arbeit wird eine allgemein anerkannte grafische Notation für Komponenten verwendet, nämlich die der UML [Obj09]. Abbildung 2-1 zeigt ein Beispiel für die Darstellung einer Komponente mit Hilfe der UML.

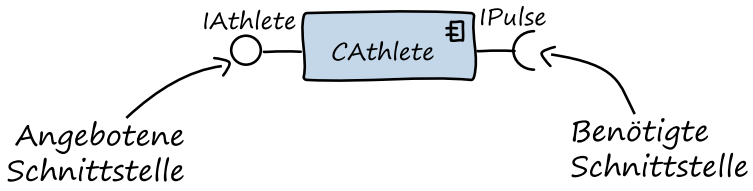


Abbildung 2-1: Beispiel für die grafische Notation einer Komponente

Eine Komponente wird hierbei als Rechteck notiert, welche einen Bezeichner für die Komponente enthält. Außerdem enthält das Rechteck oben rechts ein Symbol, welches verdeutlicht, dass es sich bei dem Rechteck um die Darstellung einer Komponente handelt. Angebotene Schnittstellen einer Komponente werden als Kreise, benötigte Schnittstellen als Halbkreise notiert. Sowohl an benötigten als auch angebotenen Schnittstellen wird zudem der Name der Schnittstelle angegeben, die angeboten bzw. benötigt wird.

Komponenteninstanzen werden zusätzlich dadurch gekennzeichnet, dass dem Komponentennamen ein Doppelpunkt vorangestellt und der Name unterstrichen wird. Optional kann vor dem Doppelpunkt ein Name für die Instanz angegeben werden. Abbildung 2-2 zeigt eine Instanz der Komponente *CAthlete*.

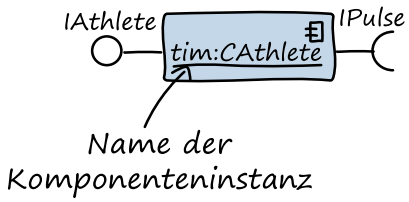


Abbildung 2-2: Beispiel für die grafische Notation einer Komponenteninstanz

Im Laufe der Arbeit wird die Notation noch um einige, lösungsspezifische Elemente erweitert. Zur Erläuterung der Problemstellung und des allgemeinen Lösungsansatzes reichen die bislang vorgestellten Komponentenbestandteile aber aus.

2.1.3 Komponentenmodelle & Komponentenframeworks

Ein Komponentenmodell legt den Rahmen für die Entwicklung und Ausführung von Komponenten fest, der strukturelle Anforderungen hinsichtlich Verknüpfungs- bzw. Kompositionsmöglichkeiten sowie verhaltensorientierte Anforderungen hinsichtlich Kollaborationsmöglichkeiten an die Komponenten stellt [GrT00]. Die Motivation, die hinter der Definition eines Komponentenmodells steht, ist den Entwicklungsrahmen und den Rahmen für die Konfiguration komponentenbasierter Anwendungen vorzugeben [GBS03]. Darüber hinaus wird zu einem Komponentenmodell meist eine Infrastruktur angeboten, die häufig benötigte, querschnittliche Mechanismen wie Verteilung, Persistenz, Nachrichtenaustausch, Sicherheit und Versionierung implementiert. Die Realisierung eines Komponentenmodells wird als Komponentensystem bzw. Komponentenplattform, oder auch als Komponentenframework

bezeichnet [HnP06]. Der Begriff der Komponente kann auch mit Hilfe des Begriffs des Komponentenmodells beschrieben werden. So wird in [HeC01] eine Komponente definiert als ein Softwareelement, das konform zu einem Komponentenmodell ist, und ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann.

Im Laufe der Zeit wurden zahlreiche Komponentenmodelle und Frameworks entwickelt, sowohl kommerzielle als auch freie. Eine der prominentesten Frameworks im Java-Umfeld sind die Enterprise Java Beans (EJB). Diese werden innerhalb eines Java-EE-Servers ausgeführt und vereinfachen die Entwicklung komplexer, mehrschichtiger verteilter Softwaresysteme. Das Komponentenmodell unterscheidet drei Arten von Komponenten, nämlich *Entity Beans*, *Session Beans* und *Message Driven Beans* [RuB10]. Diese besitzen jeweils spezielle Funktionalitäten hinsichtlich Datenhaltung, Session-Verwaltung und Nachrichtenaustausch.

Ein weiteres Komponentenmodell ist das sogenannte CORBA Component Model (CCM), welches auf CORBA 3.0 aufsetzt [Obj01]. Eine CORBA-Komponente kapselt ihren inneren Aufbau durch Schnittstellen, welche über Ports angeboten werden. Hierbei unterscheidet das Komponentenmodell unterschiedliche Port-Arten. So dienen Receptable-Ports dem Zugriff einer Komponente auf Funktionalität anderer Komponenten. Event-Source-Ports und Event-Sink-Ports realisieren den Empfang und das Senden von Nachrichten, und mit Hilfe von Stream-Source-Ports und Stream-Sink-Ports können Streams gesendet und empfangen werden.

Die OSGi Alliance (Open Services Gateway initiative) spezifiziert eine hardwareunabhängige dynamische Softwareplattform, die es erleichtert, Anwendungen und ihre Dienste zu modularisieren und zu verwalten [Osg11]. In OSGi werden Softwarekomponenten als *Bundles* bezeichnet. Diese Bundles kommunizieren über sogenannte *Services* miteinander. Das Framework zeichnet sich speziell dadurch aus, dass es Mechanismen bereitstellt, um auf den Ausfall und auf das Hinzukommen neuer Komponenten zur Laufzeit zu reagieren. Außerdem existieren Implementierungen dieses Frameworks, welche auch auf leistungsschwachen Geräten ausgeführt werden können, und somit auch für den Einsatz in mobilen und eingebetteten Systemen geeignet sind [ReA07, Mak12].

Weitere Frameworks, die nicht unmittelbar ein Komponentenframework darstellen, aber häufig als Basis zur Realisierung komponentenbasierter Anwendungen herangezogen werden, sind im Java-Umfeld RMI und Jini. RMI steht für Remote Method Invocation und ermöglicht die Realisierung verteilter Anwendungen in Java [Abt07]. Hierbei muss sich der Entwickler nicht mit der Realisierung der Netzwerkkommunikation auseinandersetzen, sondern kann Methoden eines entfernten Objektes aufrufen, als würden sie in derselben virtuellen Maschine ausgeführt werden, in der auch der Aufrufer ausgeführt wird. Jini ist ebenfalls ein Framework zur Programmierung verteilter Anwendungen, welches u.a. flexible Mechanismen zur Auffindung von Diensten in einem Netzwerk bietet [Edw00]. Des Weiteren erlaubt Jini, sowohl Daten als auch Programme über ein Netzwerk zu übertragen, und auch einen Mechanismus zur Reaktion auf den unerwarteten Ausfall einzelner Dienste.

Weitere Komponentenframeworks liefert Microsoft mit .NET [Wen12] und dem darin enthaltenen Managed Extensibility Framework (MEF) [Dob12]. MEF bietet Konzepte an, um Anwendungen modular aufzubauen und diese auch nach Auslieferung zu erweitern oder zu verändern. Als weitere Komponententechnologien von Microsoft sind COM und DCOM zu nennen [Kos00].

Auch im wissenschaftlichen Umfeld wurden im Laufe der vergangenen Jahre zahlreiche Komponentenmodelle und Frameworks entwickelt. Hierbei besitzt jedes dieser Modelle unterschiedliche Schwerpunkte, angefangen von formalen Modellen zur Beschreibung von Komponenteninstanzen und deren Verhalten [Rau01, Rau07] bis hin zu Modellen, welche die Möglichkeit zur Bewertung nichtfunktionaler Eigenschaften eines Systems noch vor dessen Implementierung ermöglichen [KrR08]. Andere Modelle realisieren zudem das Konzept der Subkomponenten [HnP06, HPM+05, SRG+08]. Diese Modelle ermöglichen die Komposition einer Komponente durch eine oder mehrere Subkomponenten. Zahlreiche dieser Modelle wurden im Rahmen eines internationalen Workshops miteinander verglichen, indem ein vorgegebenes Beispiel mit unterschiedlichen Komponentenmodellen modelliert wurde. Sowohl das Beispiel als auch die unterschiedlichen Komponentenmodelle werden in [RRM+08] ausführlich beschrieben.

2.1.4 Serviceorientierte Architekturen

Eine serviceorientierte Architektur (SOA) ist ein Architekturmuster aus dem Bereich der verteilten Systeme. Innerhalb einer solchen Architektur werden Funktionalitäten durch Dienste angeboten, und Nutzer können auf diese Dienste zugreifen, um ihrerseits bestimmte Funktionalitäten zu realisieren. Auf den ersten Blick scheinen zwischen serviceorientierten Anwendungen und komponentenbasierten Anwendungen keinerlei Unterschiede zu bestehen. Beide besitzen wohldefinierte Schnittstellen und die innere Struktur von Dienstanbietern bleibt Verwendern verborgen. Außerdem haben beide Ansätze zum Ziel, in unterschiedlichen Ausführungsumgebungen ohne Änderung wiederverwendet werden zu können. Und in der Tat ist es häufig so, dass SOA-Konzepte und komponentenbasierte Softwareentwicklung meist gemeinsam eingesetzt werden, wie am Ende des Abschnitts noch kurz erläutert wird.

Innerhalb einer serviceorientierten Architektur werden drei Rollen unterschieden, nämlich Dienstanbieter, Dienstnutzer und Dienstverzeichnis [Pap03, W3c04]. Abbildung 2-3 stellt die Beziehungen zwischen diesen Rollen dar.

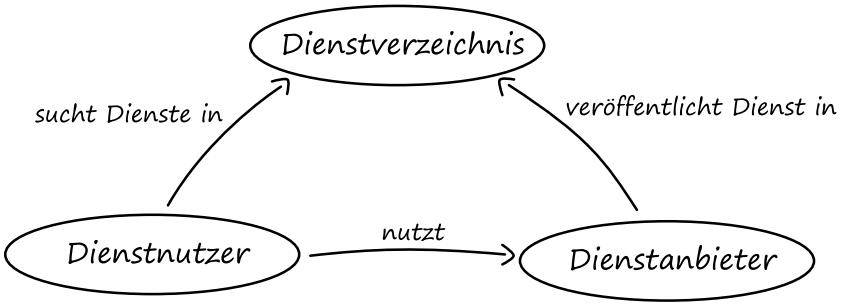


Abbildung 2-3: Rollen einer serviceorientierten Architektur

Jeder Dienst besteht hierbei aus zwei Teilen, nämlich einer Dienstimplementierung und einer Dienstbeschreibung. Ein Dienstanbieter veröffentlicht die Dienstbeschreibung innerhalb eines Dienstverzeichnisses. Diese Beschreibung enthält meist Informationen darüber, wie auf den Dienst zugegriffen werden kann sowie Informationen zu nichtfunktionalen Eigenschaften der Implementierung.

Ein potentieller Dienstnutzer sucht innerhalb des Dienstverzeichnisses anhand bestimmter Kriterien nach geeigneten Diensten. Wurde ein geeigneter Dienst gefunden, so kann der Dienstnutzer mit Hilfe der im Dienstverzeichnis hinterlegten Informationen den Dienst nutzen.

Eines der Hauptziele von SOA ist die Möglichkeit des dynamischen Bindens von Diensten. Unter dynamischem Binden versteht man den Prozess des Verbindens von Dienstanbietern mit Dienstnutzern zur Laufzeit [MRP+07]. Im Idealfall wird dieser Prozess von der Infrastruktur übernommen und bleibt sowohl den Dienstanbietern als auch den Dienstnutzern verborgen. Eine bekannte Realisierung einer solchen Architektur sind Webservices [Mel10].

Eine serviceorientierte Architektur lässt sich mit klassischen komponentenbasierten Systemen umsetzen bzw. kombinieren. Hierbei bieten Komponenten ihre Funktionalität in Form von Diensten an. Die Dienstbeschreibung besteht in der einfachsten Form aus der Angabe der Schnittstelle, welche durch den Dienst implementiert wird. Das Dienstverzeichnis ist meist durch eine sogenannte Registry realisiert. Insbesondere in dynamisch-adaptiven Systemen wird häufig eine serviceorientierte Architektur gemeinsam mit komponentenbasierter Entwicklung eingesetzt.

2.2 Grundlagen selbstorganisierender Softwaresysteme

Unter selbstorganisierenden Softwaresystemen versteht man Softwaresysteme, die in der Lage sind, selbständig auf äußere Einflüsse zu reagieren. Häufig werden derartige Systeme auch als dynamisch-adaptiv bezeichnet, oder auch unter dem Begriff des sogenannten Autonomic Computing zusammengefasst. Ein solches System zeichnet sich insbesondere dadurch aus, dass es ein besonderes Anpassungsvermögen an seine Umgebung besitzt, und die Möglichkeit hat, auf deren Veränderungen automatisiert und zur Laufzeit zu reagieren [Lad97, CCS09]. Dabei spielen Rückkopplungsschleifen, Emergenz und Selbstorganisation eine hervorgehobene Rolle.

In diesem Abschnitt wird eine Einführung in diese Art von Systemen gegeben. Die hier beschriebenen Grundlagen bilden die Basis der später vorgestellten Lösung und sind zudem für das Verständnis der Problemstellung relevant.

Die Gründe für die Realisierung eines Softwaresystems als selbstorganisierendes System sind vielfältig. Einer der Gründe ist die wachsende Komplexität heutiger Anwendungen. Die Wartung dieser Systeme ist zum Teil derart aufwändig und fehleranfällig geworden, dass im Laufe der Jahre unterschiedliche Ansätze zur Lösung des Problems entwickelt wurden. Hierzu hat IBM im Jahr 2001 das Konzept des Autonomic Computing vorgestellt [Hor01, GaC03]. Die Idee ist hierbei, dass sich das System selbständig und möglichst ohne die Notwendigkeit eines Eingriffs durch Nutzer oder Administratoren wartet und ggf. selbständig auf äußere Einflüsse oder beispielsweise auf einen Fehlerfall reagiert. Ein weiterer Grund für die Realisierung eines Softwaresystems mit der Fähigkeit sich selbst zu organisieren sind Anforderungen, die sich aus der immer größeren Verbreitung und Vernetzung kleiner, mobiler Rechner, Sensoren und Aktuatoren ergeben. So sind im Verlaufe des technologischen Fortschritts sogenannte kontextsensitive Softwaresysteme immer wichtiger geworden [RJS+06]. Die spezifische Eigenschaft dieser Systeme ist es, dass sie sich automatisch an wechselnde Ausführungsumgebungen anpassen können. So könnte eine kontextsensitive Telefon-App für das Smartphone automatisch auf ein lautloses Profil umschalten, sobald sich der Anwender in einer Sitzung befindet. In diesem Fall ist die Fähigkeit zur Selbstorganisation wesentlicher Teil

der Softwaresystem-Funktionalität. Eingesetzt werden können derartige Systeme auch, wenn Anwendungen häufig von einem Gerät auf ein anderes migriert werden müssen oder sollen. Da auf Quell- und Zielgerät ggf. unterschiedliche Ressourcen, Sensoren und Komponenten zur Verfügung stehen, ist auch hier der Einsatz selbstorganisierender Systeme durchaus sinnvoll [Pat11]. Des Weiteren bietet sich der Einsatz selbstorganisierender Softwaresysteme immer dann an, wenn von unvorhergesehenen Fluktuationen bei zur Verfügung stehenden Ressourcen, Änderungen des Kontextes (Zeit, Ort, …) oder der Nutzeranforderungen und Präferenzen [AEP+07] auszugehen ist. Selbstorganisierende Softwaresysteme besitzen einige für sie ganz spezifische Eigenschaften, welche nun kurz erläutert werden.

Eine der wesentlichen Eigenschaften eines selbstorganisierenden Softwaresystems ist die Fähigkeit zur Rekonfiguration, während die Anwendung ausgeführt wird. Derartige Systeme werden auch als dynamisch-adaptive Systeme bezeichnet [HnP06, NGM+08]. Ursprünglich rührte der Bedarf an Systemen, die sich ohne Unterbrechung des Programmablaufs anpassen lassen daher, dass die Bedeutung der Verfügbarkeit von Anwendungen gestiegen ist. Erste Ansätze zur Realisierung dynamisch-adaptiver Anwendungen wurden bereits 1985 erforscht [KrM85].

Eine weitere zentrale Eigenschaft selbstorganisierender Systeme ist die Fähigkeit, notwendige Anpassungen zum einen automatisiert zu erkennen, und zum anderen auch automatisiert umzusetzen. Hierzu realisiert ein selbstorganisierendes System meist eine sogenannte Feedbackschleife, in der der Systemkontext beobachtet und analysiert wird sowie Maßnahmen geplant und umgesetzt werden. Konkrete Ansätze hierzu werden im folgenden Abschnitt vorgestellt.

Je nach Einsatzzweck werden mit Hilfe der automatisierten Anpassung unterschiedliche Ziele verfolgt. Hierbei können u.a. folgende Ziele unterschieden werden:

- Selbstheilung
- Selbstoptimierung
- Selbstschutz
- Selbstkonfiguration

Zu den zwei wesentlichen Einsatzgebieten dynamisch-adaptiver Systeme zählen die bereits in der Einleitung erwähnten kontextsensitiven Anwendungen sowie Anwendungen im Bereich von Ambient Intelligence, Ubiquitous Computing und Pervasive Computing. Kontextsensitive Anwendungen sind in der Lage, sich selbständig an wechselnde Umgebungsbedingungen anzupassen [SAW94, Sch93, SAW94]. Insbesondere mit dem Aufkommen mobiler Anwendungen stieg der Bedarf an derartiger Software rapide.

In der Literatur existieren zahlreiche Definitionen dafür, was unter Kontext im Zusammenhang mit kontextsensitiven Systemen zu verstehen ist. In [AEP+07] wird jede Information als Kontext bezeichnet, welche in Bezug zur Interaktion zwischen Anwender und Applikation steht. Es werden hierbei drei Arten von Kontext unterschieden, nämlich der *computing context*, *physical context* und *user context*. Erstere fasst diejenigen Informationen zusammen, welche den Zustand des Rechners betreffen auf dem die Anwendung ausgeführt wird, wie z.B. Batteriekapazität und verfügbarer Speicher. Unter *physical context* werden Informationen verstanden, die die physikalischen Eigenschaften der Interaktion zwischen Nutzer und Anwendung betreffen, wie

z.B. die geographische Position, Temperatur oder Wetter. Und der *user context* beschreibt schließlich Informationen über den Nutzer sowie dessen aktuelle Tätigkeit (schläft er, fährt er Auto, sitzt er in einer Vorlesung). Auch Informationen über die aktuelle Stimmungslage werden hierunter zusammengefasst.

Neuen Schub bekam die Forschung im Bereich der selbstorganisierenden Systeme mit dem Aufkommen von *Ubiquitous Computing, Pervasive Computing* und *Ambient Intelligence* [Wei91, Wei93]. Hier wird das Ziel verfolgt, Anwendungen möglichst unauffällig in den Alltag der Menschen zu integrieren. Diese Anwendungen sollen sich zudem automatisch den Anforderungen der Anwender anpassen und ihn so im Alltag unterstützen.

Der Begriff des *Ubiquitous Computing* wurde von Mark Weiser in [Wei91] geprägt. Die beschriebene Vision war die, dass der Desktop-PC als Gerät verschwindet und durch „intelligente Gegenstände" ersetzt wird. Diese „intelligenten Gegenstände" sollen Menschen unterstützen ohne abzulenken oder aufzufallen. *Pervasive Computing* strebt im Grunde das gleiche Ziel an. Dieser Begriff wurde allerdings mehr durch die Industrie getrieben.

Das Forschungsfeld *Ambient Intelligence* (Umgebungsintelligenz) ist ebenfalls verwandt mit dem *Ubiquitous Computing* bzw. *Pervasive Computing*. Bei *Ambient Intelligence* geht es vorrangig um die massive Vernetzung von Sensoren, Recheneinheiten und Aktuatoren zur Unterstützung des Menschen im Alltag. Ein Anwendungsgebiet ist z.B. das intelligente Haus, welches in der Lage ist, sich automatisch auf die Bedürfnisse der Bewohner einzustellen und auf Ereignisse, wie z.B. das Betreten eines Raumes, angemessen zu reagieren. Die Technologie findet ebenfalls im Bereich des betreuten Wohnens Anwendung.

Ein weiteres Anwendungsfeld für selbstorganisierende Systeme findet sich im Bereich des Managements von Softwareevolution. Software muss stetig den Anforderungen der Nutzer angepasst werden, damit der Nutzen erhalten bleibt. Doch gerade in komplexen Systemen fällt eine Anpassung schwer. Die Gründe hierfür sind vielfältig. So können in großen und komplexen Systemen Auswirkungen einer Änderung nur schwer abgeschätzt werden. Die Systeme sind zudem in der Regel derart komplex, dass ein vollständiges Verständnis meist nicht erzielt werden kann. Und schließlich ist die Dokumentation des Systems meist bereits nach den ersten Änderungen derart veraltet, dass sie nicht mehr aussagekräftig ist. Das Ziel des Einsatzes selbstorganisierender Ansätze ist es, die notwendigen Anpassungen möglichst zu automatisieren und somit dem System sowohl die Erkennung eines Anpassungsbedarfs als auch dessen Umsetzung zu überlassen. Die Systemreaktionen sollen langfristig die Systemintegrität wahren und sich dabei an globalen/organisatorischen Zielen orientieren [Bos04].

2.3 Feedbackschleife in selbstorganisierenden Softwaresystemen

Basis der meisten selbstorganisierenden Softwaresysteme ist eine Feedbackschleife, in der Änderungen in der Systemumgebung beobachtet und analysiert werden sowie geeignete Anpassungsmaßnahmen geplant und umgesetzt werden [AHE+06]. Zur Realisierung der Vision selbstorganisierender Software hat IBM ein Referenzmodell für derartige Softwaresysteme entwickelt, an dem sich zahlreiche Lösungsansätze orientieren [Ibm05]. Dieses Referenzmodell ist in Abbildung 2-4 dargestellt.

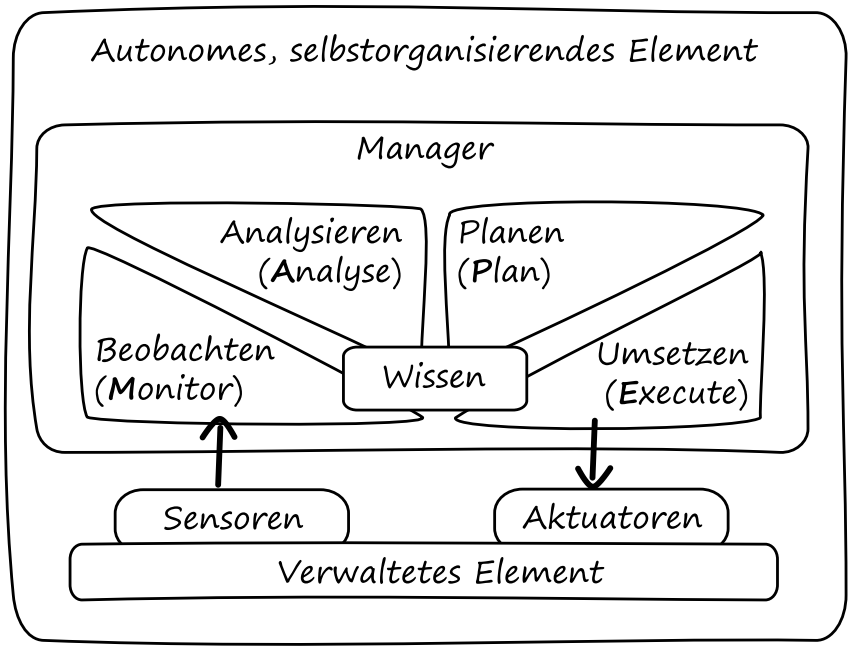


Abbildung 2-4: Referenzmodell für selbstorganisierende Softwaresysteme

Das autonome, bzw. selbstorganisierende Element ist jenes, welches selbständig auf äußere Einflüsse reagieren soll. Hierbei kann es sich sowohl um einzelne Komponenten, Sensoren, Aktuatoren oder Ähnliches handeln, als auch um komplexe Systeme.

Das autonome, selbstorganisierende Element hat zwei Bestandteile, nämlich zum einen das zu organisierende/verwaltende Element, und zum anderen einen Manager, der für die Realisierung der zuvor beschriebenen Eigenschaften selbstorganisierender Elemente verantwortlich ist (Selbstkonfiguration, Selbstheilung, Selbstoptimierung, …). Hierbei sind in der Praxis unterschiedliche Umsetzungen anzutreffen. Handelt es sich beim autonomen Element beispielsweise um ein komplexes System aus interagierenden Komponenten, so könnte eine Realisierung mit einem zentralen Manager umgesetzt werden, der für sämtliche zu verwaltenden Elemente (z.B. Komponenten) verantwortlich ist. Auf der anderen Seite ist eine Realisierung denkbar, bei der beispielsweise jede Komponente ein autonomes Element darstellt, welches jeweils einen eigenen Manager besitzt, der wiederum nur eine Komponente verwaltet.

Der Manager hat die Aufgabe, die typischen Eigenschaften selbstorganisierender Softwaresysteme für das verwaltete Element umzusetzen. Hierzu realisiert er die sogenannte MAPE-K-Schleife (**M**onitor, **A**nalyse, **P**lan, **E**xecute sowie **K**nowledge). Die Aufgaben und Realisierungsformen der einzelnen Elemente des Referenzmodells inklusive der MAPE-K-Schleife werden nun in den folgenden Abschnitten vorgestellt.

2.3.1 Manager eines selbstorganisierenden Elements

Die Managereinheit des selbstorganisierenden Elements ist verantwortlich für die Realisierung von Aufgaben der Selbstorganisation. Das Referenzmodell sieht durch Einführung einer solchen Managementeinheit eine Trennung zwischen fachlicher Funktionalität und Realisierung von Selbstorganisation vor, wobei die fachliche Funktionalität durch das verwaltete Element im Modell repräsentiert wird.

Bei der Umsetzung einer solchen Managereinheit kann zwischen reaktiv arbeitenden Managern, proaktiv arbeitenden Managern sowie hybriden Lösungen [PCH+12, AsZ12] unterschieden werden. Im ersten Fall reagiert das System erst dann auf Änderungen der Ausführungsumgebung, wenn diese bereits passiert sind. Proaktive Systeme hingegen versuchen, bestimmte Ereignisse vorherzusehen und rechtzeitig ggf. auftretenden Problemen entgegenzuwirken.

Gerade in komponentenbasierten Systemen ist diese Managereinheit auch unter dem Begriff Konfigurationseinheit bekannt. Dieser ist hier meist in der Lage, Konfigurationsdateien einzulesen, welche das gewünschte Konfigurationsverhalten beschreiben, und dieses Verhalten entsprechend umzusetzen.

2.3.2 Beobachten und Analysieren

Die Managereinheit ist verantwortlich dafür, auf äußere Einflüsse angemessen zu reagieren. Handelt es sich bei einem selbstorganisierenden Element beispielsweise um eine Komponente, so muss die Managereinheit z.B. in der Lage sein, auf den Ausfall einer verwendeten Komponente geeignet zu reagieren, sei es durch Ersetzen der Komponente oder auch durch Stoppen des verwalteten Elements, also der eigenen Komponente. Weitere häufig relevanten Ereignisse können sein [AEP+07]:

- Migration der Benutzungsschnittstelle
- Erweiterung/Veränderung der Funktionalität
- Netzwerk-Verfügbarkeit
- Anpassung der Nutzungsschnittstelle an wechselnde Bedürfnisse
- Anpassung der Qualität von Daten
- Anpassung von Sicherheitsaspekten
- Ändern des Ausführungsmodus von Anwendungen

Die Managereinheit greift zu diesem Zweck auf zur Verfügung stehende Sensoren zu. Bei diesen Sensoren kann es sich um Sensoren im klassischen Sinn handeln, wie z.B. Temperatur- oder Wasserstandssensoren, aber auch um einen Service-Registry-Listener, welcher den Wegfall oder das Hinzukommen von Komponenten registriert und dem Manager meldet.

Auslöser für Anpassungen können zudem nicht nur einzelne Ereignisse sein, sondern ggf. auch Ereignisketten oder Ergebnisse komplexer Berechnungen auf Basis einzelner Sensorinformationen. Diese Auswertung wird in der zweiten Phase der MAPE-K-Schleife durchgeführt. Die Ergebnisse der Analysephase gehen in die Planungsphase ein, um ggf. über Anpassungsmaßnahmen zu entscheiden und diese umzusetzen.

2.3.3 Planen und Umsetzen

Basierend auf den ausgewerteten Sensordaten entscheidet die Managerkomponente darüber, ob Anpassungsmaßnahmen durchgeführt werden müssen. Besteht ein Bedarf für eine Anpassung, so muss der Manager für die Umsetzung sorgen und hierzu das verwaltete Element entsprechend rekonfigurieren.

Ein verbreiteter Ansatz zur Realisierung der Planungs- und Umsetzungsphase ist der Einsatz sogenannter Event-Condition-Action-Regeln (Ereignis-Bedingung-Aktion). Tritt ein bestimmtes

Ereignis ein (z.B. das Hinzukommen einer neuen Komponente), wird anschließend geprüft, ob die Bedingung der Regel erfüllt ist (z.B. wird geprüft, ob die neue Komponente für eine Verwendung in Frage kommt). Ist dies der Fall, so wird die Aktion, welche in der Regel hinterlegt ist, ausgeführt (z.B. die Verwendung der Komponente). Diese Regeln werden meist von einem Administrator oder Anwendungsentwickler beim Managerelement des selbstorganisierenden Elements hinterlegt und durch dieses realisiert. Im Verlauf der vergangenen Jahre wurden zahlreiche Sprachen zur Spezifikation derartiger Regeln entwickelt und zur Realisierung selbstorganisierender Systeme verwendet, wie z.B. Ponder [DDL+00, LLS03], PDL [LBN99] und weitere [ACG+05, BBC+02, LMK+01, PYP02]. Der Einsatz von Event-Condition-Action-Regeln birgt allerdings auch Nachteile. So kann die Anzahl der Regeln in komplexen Systemen schnell sehr groß und somit unübersichtlich werden. Außerdem lassen sich zur Designzeit nicht alle widersprüchlichen Regeln erkennen, so dass zur Laufzeit ggf. nicht eindeutig automatisch entschieden werden kann, welche Aktionen durchzuführen sind [LuS97].

Häufig wird der zuvor vorgestellte Ansatz kombiniert mit einem sogenannten architekturbasiertem Ansatz. Bei diesem Ansatz wird zunächst die gewünschte Architektur des Systems modelliert, inklusive bestimmter Anforderungen und Ziele. Diese Architektur wird derart spezifiziert, dass das Managerelement dieses interpretieren kann. Das Managerelement hat dann zur Laufzeit die Aufgabe sicherzustellen, dass die Vorgaben an die Architektur eingehalten werden. Dieser Ansatz wird meist verknüpft mit dem Einsatz von Event-Condition-Action-Regeln. In diesem Fall werden beispielsweise Regeln hinterlegt, für den Fall, dass eine Verletzung der vorgegebenen Architektur vorliegt. Die in dieser Arbeit vorgestellte Lösung basiert auf einem architekturbasiertem Ansatz. Verwandte Ansätze in diesem Forschungsgebiet werden in dieser Arbeit in Abschnitt 3.3 vorgestellt, nachdem die Problemstellung und ein Anwendungsbeispiel erläutert wurden.

Ein weiterer Ansatz zur Realisierung der Planungs- und Entscheidungsphase ist die Verwendung sogenannter *goal policies*. Hierbei werden vom Entwickler lediglich die Ziele und Randbedingungen der Anwendung definiert, während die konkrete Umsetzung einem Framework überlassen wird. Ein Framework, welches diesen Ansatz verfolgt ist *Unity* [CSW+04, WTK+04].

Wurde der Bedarf zur Anpassung erkannt, muss anschließend eine entsprechende Rekonfiguration des Systems erfolgen. Die Möglichkeiten, die einzelne Frameworks bieten, sind sehr unterschiedlich. Hierbei werden in der Regel zwei Kategorien unterschieden, nämlich Parameteradaption und kompositionelle Adaptation [MSK+04]. Bei der Parameteradaption werden bestimmte Variableninhalte des Programms durch die Managerkomponente (siehe Abbildung 2-4) verändert, welche zu einer Änderung des Verhaltens führt. Zahlreiche kontextsensitive Anwendungen setzen diese Form der Anpassung ein [SoG02, SDA99, FKS97, DeP00]. Ein Nachteil dieses Ansatzes ist, dass lediglich solche Änderungen vorgenommen werden können, die bereits zur Entwicklungszeit explizit vorgesehen wurden. Bei der kompositionellen Anpassung wird die Struktur der Anwendung verändert, also die Menge der verwendeten Komponenten sowie die Verbindungen zwischen diesen. Hierbei können auch noch zur Laufzeit beispielsweise neue Algorithmen hinzugefügt werden [HiS96, AkC03, CHS02]. Bei kompositioneller Adaption kann zusätzlich unterschieden werden zwischen Systemen, bei denen die Komponententypen zur Entwicklungszeit bekannt sein müssen und jenen, bei denen diese Voraussetzung nicht erfüllt sein muss [CFB01]. Bei ersterer Variante können in den Event-Condition-Action-Rules konkrete Komponenten bzw. Komponententypen referenziert werden

und so z.B. eine bestimmte Instanz einer Komponente erzeugt werden, falls eine andere ausgefallen ist. In der zweiten Variante ist es zusätzlich möglich, zur Laufzeit neue Komponententypen in das System zu integrieren. Im Bereich der kompositionellen Adaption werden meist fünf elementare Operationen zur Realisierung dynamischer Rekonfiguration betrachtet [HnP06]:

1. Entfernen einer Komponente
2. Hinzufügen einer Komponente
3. Entfernen einer Verbindung
4. Hinzufügen einer Verbindung
5. Hinzufügen/Entfernen von Komponentenschnittstellen.

In dieser Arbeit wird ein Ansatz vorgestellt, welcher die zweite Variante realisiert, um so insbesondere den Anforderungen mobiler dynamisch-adaptiver Anwendungen gerecht werden zu können. Hierbei werden die aufgeführten elementaren Operationen unterstützt.

2.3.4 Wissensbasis eines autonomen Elements

Damit die Managerkomponente des Referenzmodells für selbstorganisierende Systeme (siehe Abbildung 2-4) angemessen auf äußere Einflüsse reagieren kann, benötigt sie bestimmte Informationen. Hierzu gehören insbesondere Informationen über den aktuellen Zustand des Systems, Informationen über die Umgebung sowie Informationen darüber, welche Ziele bei der Rekonfiguration angestrebt werden sollen.

Um hier automatisiert eine optimale Entscheidung über die zu treffenden Rekonfigurationsmaßnahmen zu fällen, werden meist sogenannten *utility functions* verwendet [AHE+06, AEP+07, WTK+04]. Ziel dieser *utility functions* ist es, die Qualität einer Anwendungskonfiguration auszudrücken. Hierzu werden der Funktion alle relevanten Entscheidungsparameter übergeben (Nutzerpräferenz, Ausführungskontext usw.). Als Ergebnis liefert die Funktion meist eine reelle Zahl zurück, welche die Qualität einer Konfiguration wiederspiegelt.

Ziel dieser Phase ist es, unter unterschiedlichen möglichen Anwendungskonfigurationen diejenige auszuwählen, welche sowohl aus Nutzersicht als auch aus Systemsicht optimal ist. Bei der Entscheidung darüber, auf welche Art und Weise die Anwendung rekonfiguriert werden soll, spielen in der Regel zahlreiche Faktoren eine Rolle. Hierzu zählen u.a. die aktuell zur Verfügung stehenden Ressourcen, der aktuelle Ausführungskontext, Nutzerpräferenzen sowie allgemeine Vorgaben, welche durch den Anwendungsentwickler definiert wurden.

2.4 Frameworks für selbstorganisierende Softwaresysteme

Eine der ersten Plattformen für verteilte dynamisch-adaptive Anwendungen war CONIC. In CONIC bestehen Anwendungen aus interagierenden Modulen, und die Konfiguration der Anwendung wurde durch einen zentralen Konfigurationsmanager durchgeführt. Dieser war in der Lage, Befehle vom Administrator entgegenzunehmen, mit deren Hilfe die Struktur der Anwendung festgelegt und verändert werden konnte. CONIC definiert hierzu eine eigene Konfigurationssprache [MKS89]. Sie enthält u.a. Befehle zur Erzeugung von Komponenteninstanzen sowie zur Definition von Verbindungen zwischen diesen. Zu den frühen Plattformen, welche dynamische Rekonfiguration unterstützt haben, gehört auch REX [Kra90,

KMS+92]. Genau wie in CONIC werden hier verteilte und parallele Systeme als miteinander verbundene Komponenteninstanzen betrachtet. Schnittstellen werden in einer eigens definierten Schnittstellenbeschreibungssprache definiert, während die Funktionalität selber in beliebigen Programmiersprachen realisiert werden kann. Bei Komponenten handelt es sich in REX um Typen, von denen zu einem Zeitpunkt mehrere Instanzen existieren können. Die Menge von Instanzen sowie deren Verbindungen werden in einer separaten Konfigurationssprache namens Darwin definiert. Die Sprache bietet spezielle Befehle zur Rekonfiguration einer laufenden Anwendung an. So kann zur Laufzeit sowohl die Menge der Instanzen verändert werden, als auch die Verbindungen zwischen diesen Instanzen.

Wie bereits erwähnt, werden Rekonfigurationen in CONIC und REX durch einzelne Befehle spezifiziert, welche die Rekonfiguration durchführen. Soll beispielsweise eine Komponente aus der Anwendung entfernt werden, wird mit Hilfe des Befehls *remove* die Komponente entfernt und mit *unlink* von anderen Komponenten getrennt. Dieses Vorgehen ist allerdings sehr aufwändig. Denn für jede Rekonfiguration muss im Grunde ein eigenes Rekonfigurationsprogramm geschrieben werden, in welchem die einzelnen notwendigen Schritte angegeben sind. Der Austausch einer Komponente kann dazu führen, dass ein großes Rekonfigurationsprogramm entwickelt werden muss. In [WaS95] wurde deshalb ein Ansatz vorgestellt, welches von der Notwendigkeit der Angabe einzelner Rekonfigurationsbefehle abstrahiert. Hier werden vielmehr bestimmte Anwendungskonfigurationen vorgegeben und je nach Situation vom System umgesetzt. Es handelt sich hierbei also um eine deklarative Art der Rekonfigurationsspezifikation.

Eine weitere Infrastruktur zur dynamischen Rekonfiguration komponentenbasierter Anwendungen ist DAiSI (Dynamic Adaptive System Infrastructure). Diese unterstützt zum einen das automatische Hinzufügen und Entfernen von Komponenten einer laufenden Anwendung, und zum anderen unterschiedliche Ausführungsmodi einzelner Komponenten [NKA+07].

Eine besondere Art dynamisch-adaptiver Systeme stellen solche dar, bei denen zur Entwicklungszeit nicht alle Komponenten bekannt sind. Hierbei können somit zur Laufzeit gänzlich neue Komponenten hinzukommen und vollautomatisiert in die Anwendung integriert werden. Ist zudem die Semantik der Komponentenschnittstellen nicht eindeutig definiert, werden Ansätze benötigt, welche die Semantik von benötigten und angebotenen Schnittstellen überprüfen. Ein solcher Ansatz wurde in [Nie10] vorgestellt. Hierbei wurde mittels Testfällen bestimmt, ob eine angebotene Schnittstelle alle Anforderungen der benötigten Schnittstelle erfüllt.

Im Bereich der serviceorientierten Architekturen ist u.a. das Framework *ProAdapt* zu nennen [AsZ12]. Dieses unterstützt die Anpassung der Dienstekomposition als Reaktion auf vier Klassen von Situationen:

- Probleme, welche die Ausführung der Anwendung stoppen
- Probleme, die die Ausführung in einer nicht optimalen Konfiguration erfordern
- Auftreten neuer Anforderungen
- Bereitstellung von besser geeigneten Diensten

ProAdapt ist hierbei in der Lage, einzelne Dienste zu ersetzen, sowie ganzen Dienstkompositionen auszutauschen.

In [RaP02, RaP03] wurde ein Framework vorgestellt, welches die dynamische Rekonfiguration mobiler Anwendungen auf Basis von .NET ermöglicht. Eine Anwendung setzt sich innerhalb aus interagierenden Komponenten zusammen. Eine Anwendungskonfiguration wird innerhalb des Frameworks mit Hilfe von XML spezifiziert. Ein zentraler Konfigurationsmanager ist in der Lage, diese Spezifikation zu interpretieren und umzusetzen. Er instanziiert hierzu je nach Bedarf benötigte Komponenten und verknüpft diese untereinander gemäß der Spezifikation. Die Spezifikation einer Anwendung kann hierbei beliebig viele Konfigurationen enthalten. An diese sind Bedingungen geknüpft, unter welchen Umständen welche der Konfigurationen realisiert werden soll. Die Umgebung wird mit Hilfe spezieller *Observer*-Komponenten durch das Framework beobachtet. Jede Anwendungskomponente muss, um durch das Framework konfiguriert werden zu können, eine Schnittstelle namens *IConfigure* implementieren. Das Framework unterstützt sowohl das dynamische Hinzufügen als auch Entfernen von Komponenten und Verbindungen.

3 Problemanalyse

In diesem Kapitel wird die Problemstellung, welche in dieser Arbeit behandelt wird, detailliert erläutert. Hierzu wird zunächst ein kleines Anwendungsbeispiel vorgestellt, anhand dessen die Problemstellung im darauffolgenden Abschnitt diskutiert wird. Das Beispiel dient zudem in den darauffolgenden Kapiteln der Veranschaulichung der erarbeiteten Lösungskonzepte.

3.1 Anwendung zur Trainingsunterstützung im Biathlon

In diesem Abschnitt wird eine komponentenbasierte, verteilte mobile Anwendung vorgestellt, bei der sich die Realisierung als selbstorganisierendes Softwaresystem anbietet. Dieses Beispiel wird in den folgenden Kapiteln verwendet, um bestimmte Sachverhalte zu erläutern und die Funktionsweise der vorgestellten Lösung zu veranschaulichen. Außerdem dient das Beispiel im nächsten Abschnitt dazu, die verwandten Arbeiten in einen Kontext zu setzen.

3.1.1 Die Anwendungsdomäne

Beim hier vorgestellten Beispiel handelt es sich um eine Anwendung zur Trainingsunterstützung für Biathleten. Biathlon ist eine Kombinationssportart, die sich aus zwei Disziplinen zusammensetzt. Eine dieser Disziplinen ist der Skilanglauf, bei der im Wesentlichen die Ausdauer des Athleten entscheidend ist. Die zweite Disziplin ist das Schießen, bei der Treffsicherheit gefragt ist.

Beim Biathlon werden während eines Wettkampfes mehrere Runden gelaufen, wobei die Rundenlänge abhängig ist von der Wettkampfart und davon, ob es sich um ein Damen- oder Herrenrennen handelt. Nach Beendigung einer Laufrunde muss der Biathlet das Schießen bewältigen. Das Schießen wird auf einer Schießanlage absolviert. Eine Schießanlage besteht meist aus 30 Schießständen. Dem Sportler wird beim Einlauf in die Schießanlage ein Schießstand zugewiesen, an dem er seine Schießübung absolvieren muss. Man unterscheidet zwischen Liegend- und Stehendschießen bzw. Liegend- und Stehendanschlag. In einem Wettkampf wird je nach Wettkampfart entweder einmal liegend und einmal stehend geschossen, oder zweimal liegend und zweimal stehend. Bei jeder Schießeinlage müssen fünf Scheiben getroffen werden, wobei der Scheibendurchmesser beim Liegendanschlag kleiner ist als beim Stehendanschlag. Werden eine oder mehrere Scheiben nicht getroffen, so gibt es je nach Wettkampfart unterschiedliche Strafen für den Athleten. Hier wird meist entweder pro Fehlschuss eine Zeitstrafe verhängt, oder der Läufer muss pro Fehlschuss eine sogenannte Strafrunde absolvieren. Hierzu verlässt er nach Abgabe seiner fünf Schüsse die Schießanlage, darf aber nicht sogleich die nächste Laufrunde absolvieren. Stattdessen muss er in eine Strafrunde von 150m Länge abbiegen und diese je nach Anzahl der Fehlschüsse ggf. mehrere Male durchlaufen, bevor er wieder in reguläre Laufrunde abbiegen darf.

Beim Schießen sind im Wesentlichen zwei Herausforderungen zu bewältigen. Zum einen muss man in der Lage sein, nach einer intensiven Ausdauerleistung eine ruhige Körperhaltung beim Schießen einnehmen zu können. Hierzu muss der Athlet vor Einlauf in den Schießstand sein Lauftempo rechtzeitig anpassen. Kommt er überlastet oder mit zu geringem Puls zum Schießen, beginnen während des Schießens die Arme zu zittern. Ein ähnlicher Effekt tritt auch auf, wenn er zu lange für eine Schießübung benötigt. Zum anderen muss der Biathlet die Wetterbedingungen und hier insbesondere die Windbedingungen berücksichtigen. Schon geringe Seitenwinde können die Flugbahn des Geschosses derart beeinflussen, dass das Ziel verfehlt wird. Aus diesem

Grunde können die Athleten je nach Windverhältnissen ihre Visiereinrichtung an den Wind anpassen.

Verlässt der Athlet nach seiner Schießübung und nach ggf. gelaufenen Strafrunden die Schießanlage, beginnt eine neue Laufrunde. Hier haben dann die Trainer die Möglichkeit, ihren Schützlingen Rückmeldungen zum aktuellen Wettkampfstand sowie insbesondere auch zum Schießergebnis zu geben. Pro Team befinden sich meist mehrere Trainer an der Strecke bzw. am Schießstand. Die Trainer am Schießstand sind mit Ferngläsern ausgerüstet und beobachten aus einer Coaching-Zone heraus die Schießleistung ihrer Athleten. Sie können hierbei beobachten, wo die Geschosse auf den Scheiben eingeschlagen sind. Häufig haben die Schüsse während einer Schießeinlage eine gewisse Tendenz in eine Richtung. So könnte es z.B. vorkommen, dass die Schüsse bei Wind von rechts eine Linkstendenz aufweisen. Dies kann der Athlet während des Schießens nicht immer genau erkennen. Aus diesem Grunde gibt der Trainer am Schießstand die Information über das Schießen an seine Co-Trainer an der Strecke weiter. Diese informieren dann den entsprechenden Athleten durch Zuruf oder mittels Tafeln über ggf. aufgetretene Schießtendenzen. Während des Schießens ist im Wettkampf kein Coaching durch Trainer erlaubt.

Für die softwaregestützte Trainingsoptimierung im Biathlon ergeben sich basierend auf dieser Domänenbeschreibung einige Ansätze, die nun kurz vorgestellt werden.

Die Trainer haben sowohl während des Wettkampfes, als auch während des Trainings beim Biathlon die Schwierigkeit, die Leistung aller ihrer Athleten im Blick zu behalten. Daher ist es für die Trainer auch schwierig, jedem angemessene Rückmeldungen während des Trainings zu geben. Aus diesem Grunde befinden sich insbesondere in Profiteams zahlreiche Trainer und Helfer zur Unterstützung an der Strecke. Die hier vorgestellte Anwendung unterstützt die Beteiligten dabei, das Training effektiv durchzuführen. Sie übermittelt dem Trainer bestimmte Informationen über seine Sportler und stellt diese übersichtlich auf seinem Laptop dar. Hierzu gehören u.a. die aktuelle Position des Athleten auf der Strecke, seine Geschwindigkeit, die aktuelle Steigung des Streckenteils an dem sich der Sportler befindet, sowie sein aktueller Puls.

Auch Sportler können vom softwaregestützten Training profitieren. Für sie ist u.a. wichtig, den Puls zu überwachen, um Überlastungen, aber auch Unterforderungen zu verhindern. Diese Daten, aber auch Warnungen bei Über- oder Unterschreitung von Grenzwerten, werden dem Sportler bereits während des Trainings automatisch durch eine Sportlerkomponente mitgeteilt. Diese Komponente könnte z.B. auf einem Smartphone oder in einer Armbanduhr des Sportlers realisiert werden. Per Sprachausgabe oder auf einem kleinen Display könnten dann die Informationen an den Sportler übermittelt werden. Über eine drahtlose Verbindung zum Trainer kann dieser ihm auch Hinweise zur Trainingssteuerung geben.

Während des Trainings unterstützt das System den Sportler auch beim Schießen. Hierbei kann dem Biathleten unmittelbar nach dem Schießen die Tendenz seiner Treffer mitgeteilt werden. Die entsprechenden Informationen werden von Komponenten bereitgestellt, welche an den einzelnen Schießständen installiert sind.

3.1.2 Domänenschnittstellen

Es wird für das Beispiel angenommen, dass für die Biathlondomäne einige Schnittstellen durch Domänenexperten definiert wurden. Diese werden in diesem Abschnitt kurz vorgestellt.

Abbildung 3-1 gibt eine Übersicht über die definierten Schnittstellen, welche für das Beispiel benötigt werden.

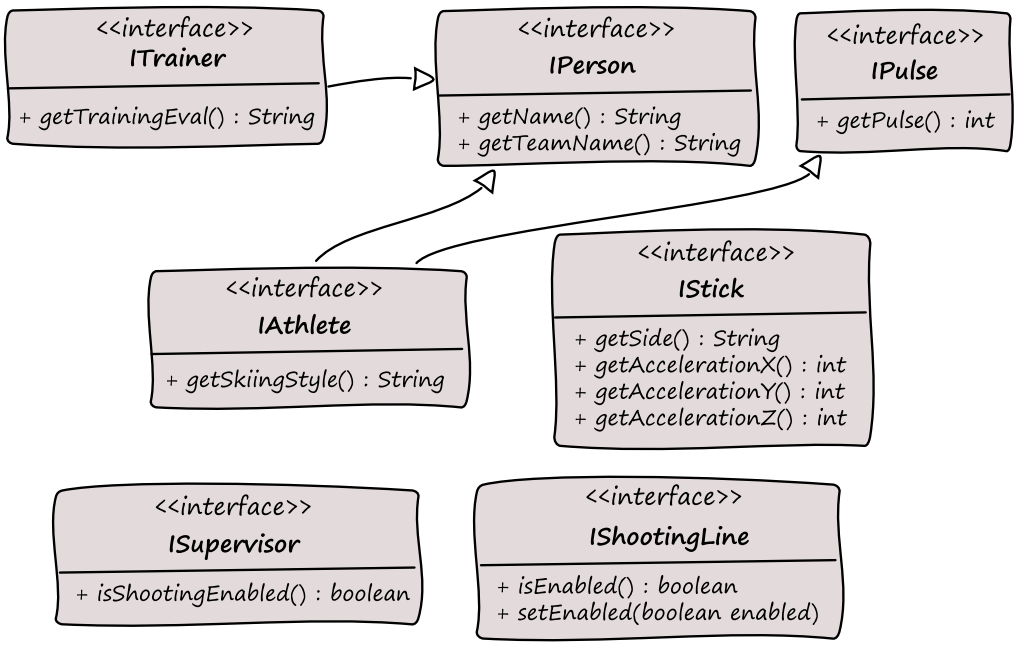


Abbildung 3-1: Domänenschnittstellen für die Biathlonanwendung

Die Schnittstellen sind in Form eines UML-Klassendiagramms [Obj09] dargestellt. Jedes Rechteck stellt eine Schnittstelle dar, dessen jeweiliger Name in der ersten Zeile angegeben ist. Zusätzlich definiert jede Schnittstelle eine Menge von Methoden inklusive Übergabeparameter und Rückgabetyp.

Für die Biathlondomäne wurde eine Schnittstelle namens *IPulse* definiert, über die der aktuelle Puls eines Sportlers abgefragt werden kann. Des Weiteren ist eine Schnittstelle namens *IStick* definiert, über die Informationen zu den Beschleunigungswerten eines Skistocks abgefragt werden können. Mit Hilfe dieser Beschleunigungswerte kann unter anderem die aktuelle Lauftechnik eines Sportlers bestimmt werden (Doppelstockschub, Diagonalschritt). Die Methode *getSide* der Schnittstelle liefert zudem die Seite zurück, auf der der Skistock gerade eingesetzt wird (also links oder rechts).

Die Schnittstelle *IShootingLine* repräsentiert eine Schießbahn, welche durch Aufruf der Methode *setEnabled* durch die Schießaufsicht für dies Schießen freigegeben oder gesperrt werden kann. Zusätzlich kann abgefragt werden, ob die Schießbahn gerade freigegeben oder gesperrt ist. Die Schießaufsicht ist durch die Schnittstelle *ISupervisor* repräsentiert, über die in Erfahrung gebracht werden kann, ob die gesamte Schießanlage für das Schießtraining freigegeben ist, oder ob sie z.B. zu Wartungszwecken gesperrt ist.

Die zwei Hauptakteure der Beispieldomäne sind Trainer und Sportler, welche durch die Schnittstellen *ITrainer* und *IAthlete* repräsentiert sind. Über die Schnittstelle *ITrainer* können im Wesentlichen Trainingsauswertungen erfragt werden, während über die Schnittstelle *IAthlete* der aktuelle Laufstil sowie der aktuelle Puls eines Athleten abgefragt werden kann. Sowohl Trainer als auch Sportler haben einen Namen sowie eine Teamzugehörigkeit. Die Methoden zur Abfrage dieser Informationen sind in der Schnittstelle *IPerson* zusammengefasst, von der sowohl die Schnittstelle *ITrainer* als auch die Schnittstelle *IAthlete* erben.

3.1.3 Komponenten der Anwendung

Im Beispiel sind zur Erfüllung der Anforderungen unterschiedliche Komponenten vorgesehen, welche zur Laufzeit miteinander interagieren. Diese implementieren bzw. benötigen die soeben vorgestellten Schnittstellen. Die Komponenten für dieses Beispiel sind in Abbildung 3-2 dargestellt werden nun kurz vorgestellt.

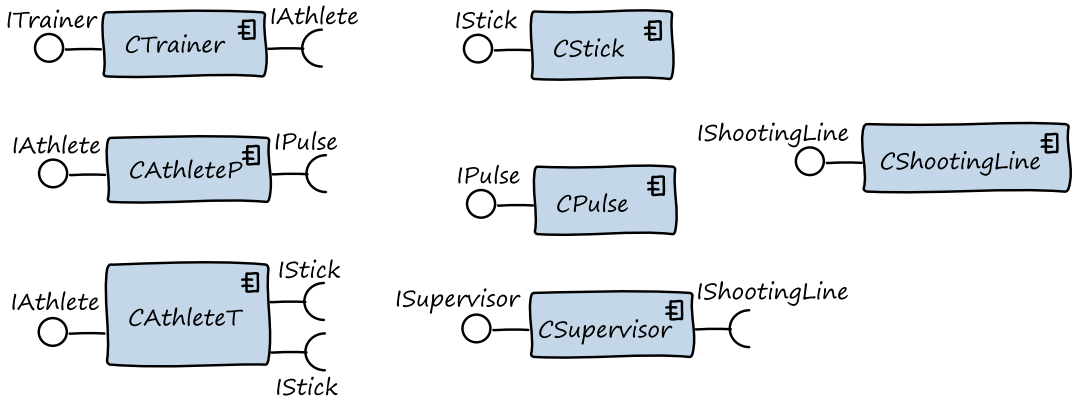


Abbildung 3-2: Die Biathlonkomponenten für das Beispiel

Die Komponente *CPulse* realisiert die Domänenschnittstelle *IPulse*, über die der aktuelle Puls eines Sportlers abgefragt werden kann. Die Komponente *CStick* implementiert die Schnittstelle *IStick*, über die u.a. die Bewegungen eines Skistocks ermittelt werden kann. Jede Instanz der Komponente *CShootingLine* realisiert die Funktionalität, welche zur Steuerung einer Schießbahn notwendig ist. So kann über die angebotene Schnittstelle der Schießstand aktiviert oder deaktiviert werden.

Die Komponente *CTrainer* realisiert die Schnittstelle *ITrainer*, über die u.a. Trainingsauswertungen abgefragt werden können. Diese können z.B. Auswertungen bzgl. des Pulsverlaufs seiner Athleten beinhalten. Zu diesem Zweck benötigt die Komponente Zugriff auf Komponenten, welche die Schnittstelle *IAthlete* implementieren. Für das Beispiel stehen zwei Athletenkomponenten zur Verfügung, nämlich *CAthleteP* und *CAthleteT*. Die Komponente *CAthleteP* wertet den Puls eines Athleten aus. Diesen Puls fragt er von einem Pulssensor ab, der die Schnittstelle *IPulse* implementiert. In diesem Beispiel wird der Puls eines Athleten von einer Instanz der Komponente *CPulse* bereitgestellt. Die Komponente *CAthleteT* hingegen analysiert die Lauftechnik eines Athleten und wertet hierzu die Stockbewegungen aus. Zu diesem Zweck benötigt die Komponente Zugriff auf zwei Skistockkomponenten, welche im Beispiel durch Instanzen der Komponente *CStick* bereitgestellt werden. Und schließlich wurde für das Beispiel die Komponente *CSupervisor* eingeführt. Diese wird von der Schießaufsicht dazu verwendet, die einzelnen Schießbahnen zu steuern und das Schießen zu überwachen. Hierzu benötigt die Komponente Zugriff auf die Schießbahninstanzen. Über die Schnittstelle *IShootingLine* kann die Schießaufsicht einzelne Schießbahnen je nach Situation aktivieren oder deaktivieren.

Eine laufende Anwendungskonfiguration besteht aus einer Menge interagierender Instanzen der hier vorgestellten Komponenten. Im folgenden Abschnitt werden nun einige Konfigurationen vorgestellt, die unter Verwendung der definierten Komponenten erzeugt werden können.

3.1.4 Konfigurationen der Anwendung

Die Anwendung, welche in den folgenden Kapiteln als Beispiel und der Problemdefinition dient, besteht aus interagierenden Instanzen derjenigen Komponenten, welche im vorherigen Abschnitt vorgestellt wurden. Es lassen sich mit Hilfe dieser Komponenten unterschiedlichste Anwendungen erzeugen. Einige beispielhafte Konfigurationen werden im Folgenden kurz vorgestellt.

Konditionstraining

Angenommen, es soll ein reines Ausdauertraining durchgeführt werden. In der einfachsten Form werden hierfür lediglich zwei Komponenten benötigt, nämlich eine Sportlerkomponente und eine Pulskomponente. Die Sportlerkomponente liest den aktuellen Puls aus der Pulskomponente aus und warnt den Sportler, falls der Puls zu hoch oder zu niedrig ist. Abbildung 3-3 zeigt die entsprechende Konfiguration für den Fall, dass lediglich ein Sportler das Training absolviert.

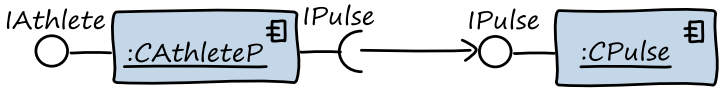


Abbildung 3-3: Einfache Anwendung zur Pulsüberwachung eines Sportlers

Sowohl für den Athleten als auch für den Pulssensor existiert jeweils genau eine Instanz der entsprechenden Komponente. Die Athletenkomponente nutzt hier die Schnittstelle *IPulse*, welche von der Pulskomponente angeboten wird.

Denkbar ist auch eine etwas komplexere Konfiguration, an der ein Trainer und mehrere Athleten mit jeweils einem Pulssensor beteiligt sind. Eine solche Konfiguration ist in Abbildung 3-4 dargestellt.

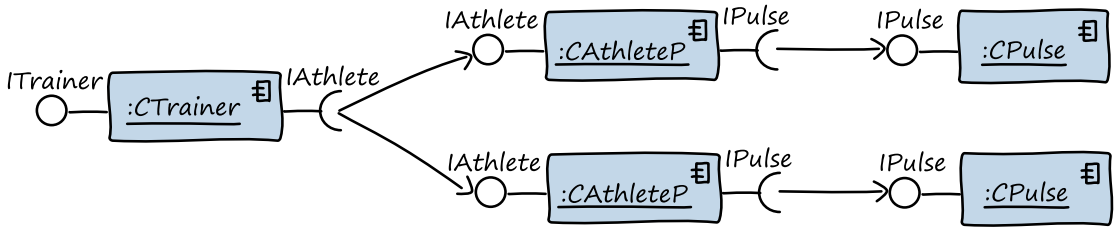


Abbildung 3-4: Konditionstraining mehrerer Athleten unter Aufsicht eines Trainers

In diesem Fall sind ein zweiter Sportler, eine zweite Pulskomponenteninstanz sowie eine Instanz einer Trainerkomponente hinzugekommen. Die Trainerkomponente kann über die Schnittstelle *IAthlete* der Sportlerkomponenten den aktuellen Puls abfragen. Der Trainer kann diese Daten auswerten und ggf. das Training entsprechend steuern.

Techniktraining

Eine weitere Menge von Anwendungskonfigurationen ergibt sich für das Training der Lauftechnik. Hierzu wurde die Komponente *CAthleteT* eingeführt. Diese liest Informationen über die Stockbewegungen zweier Skistöcke aus und bestimmt hieraus die Lauftechnik sowie dessen Qualität. Eine mögliche Konfiguration ist in Abbildung 3-5 angegeben.

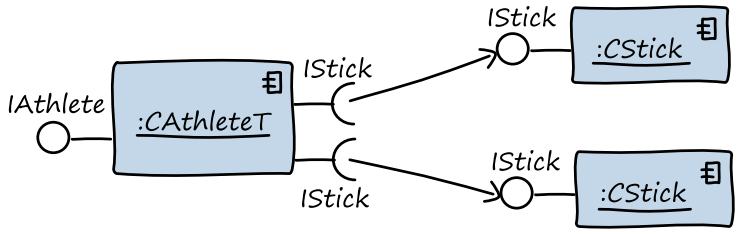


Abbildung 3-5: Konfiguration für das Techniktraining eines Athleten

Auch für das Techniktraining lassen sich beliebig komplexe Konfigurationen erzeugen. Allerdings sind auch hier einige Aspekte zu beachten, um sinnvolle Anwendungen zu erhalten. Bei Verwendung der Athletenkomponente muss z.B. sichergestellt werden, dass sie mit einem linken und einem rechten Skistock verbunden ist, da ansonsten eine aussagekräftige Analyse der Lauftechnik nicht möglich ist. Auch kann eine solche Auswertung nur stattfinden, wenn die Athletenkomponente auch wirklich mit zwei Stöcken verbunden ist.

Wettkampftraining

Auch komplexere Anwendungen können mit Hilfe der vorgestellten Komponenten erzeugt werden. Eine kleines Beispiel hierfür zeigt Abbildung 3-6.

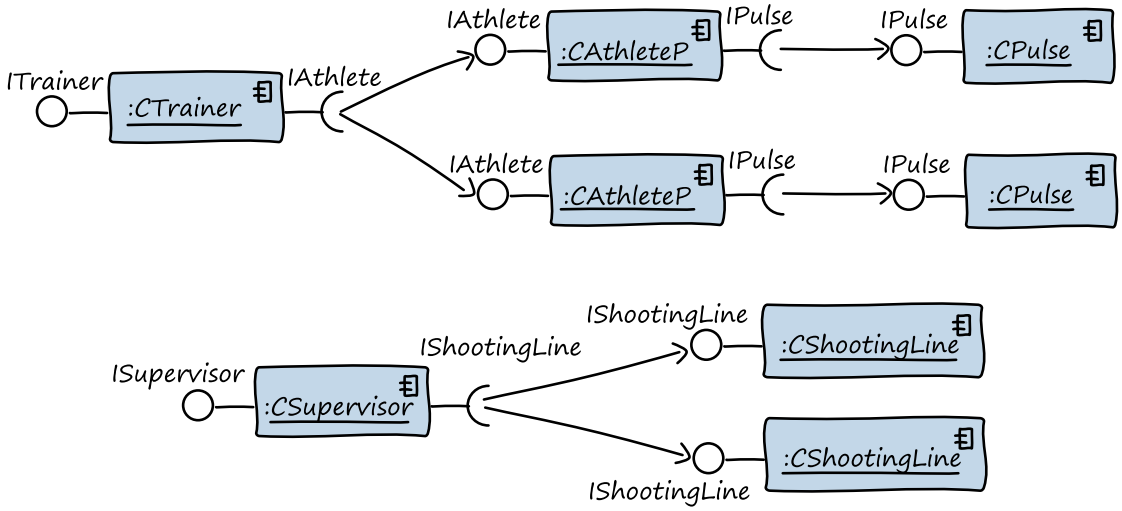


Abbildung 3-6: Konfiguration mit mehreren Trainern und einer Schießaufsicht

In diesem Fall besteht die Anwendung aus einem Trainer, welcher zwei Sportler trainiert, die jeweils mit einem Pulssensor verbunden sind. Außerdem wird in dieser Anwendung das Schießen mit einbezogen. Soll während des Trainings auch das Schießen trainiert werden, ist aus rechtlichen Gründen die Anwesenheit einer Schießaufsicht erforderlich, welche im System durch eine Instanz der Komponente *CSupervisor* repräsentiert wird. Diese Komponente steuert zwei Schießbahnen, für jeden der beiden Athleten eine.

Anhand dieser Komponenten wird nun im folgenden Abschnitt die dieser Arbeit zugrundeliegende Problemstellung im Detail vorgestellt. Im weiteren Verlauf der Arbeit wird dann ein Framework vorgestellt, welches die automatische Erzeugung gültiger Konfigurationen zur Laufzeit auf Basis vorhandener Komponenteninstanzen ermöglicht.

3.2 Problemstellung

Anhand des soeben vorgestellten Anwendungsbeispiels wird nun in diesem Abschnitt die Problemstellung beschrieben, welche dieser Arbeit zugrunde liegt. Das Hauptziel dieser Arbeit ist es, eine Infrastruktur bereitzustellen, welches die Möglichkeit bietet, auf Basis einer gegebenen Menge von Komponenteninstanzen automatisiert und zur Laufzeit Konfigurationen zu erzeugen, welche die gegebenen Anforderungen erfüllen. Hierbei sollen allerdings einige Aspekte berücksichtigt werden, welche in bestehenden Arbeiten bislang nicht gelöst wurden (siehe Abschnitt 3.3). Diese Aspekte werden im Folgenden näher erläutert.

Die Problemstellung ergibt sich aus dem Spannungsfeld zweier Ziele. Auf der einen Seite ist es, wie bereits in der Einleitung erwähnt und im Folgenden näher erläutert wird, wünschenswert, ein dynamisch-adaptives System auf Basis selbstorganisierender Komponenten zu entwickeln. Auf der anderen Seite möchte man hierbei in einigen Fällen Einfluss auf das entstehende Komponentennetzwerk ausüben können, um die Einhaltung anwendungsspezifischer Anforderungen an die Konfiguration gewährleisten zu können. Im Folgenden werden nun zunächst diese zwei Ziele sowie deren Hintergrund genauer beleuchtet und anschließend das sich daraus ergebende Spannungsfeld und die Problemstellung abgeleitet.

3.2.1 Anwendungskonfiguration auf Basis selbstorganisierender Komponenten

Es gibt viele Gründe, eine dynamisch-adaptive Anwendung auf Basis selbstorganisierender Komponenten zu entwickeln. Wie bereits in Kapitel 2 erläutert wurde, sind diese Komponenten in der Lage, Abhängigkeiten zu anderen Komponenten selbständig aufzulösen sowie auf Ereignisse in der Umgebung angemessen zu reagieren. Bei solchen Ereignissen handelt es sich üblicherweise um das Hinzukommen oder Wegfallen von Komponenten sowie Zustands- oder Kontextänderungen.

Eine dynamisch-adaptive Anwendung, welche auf selbstorganisierenden Komponenten basiert, kommt ohne zentrale Konfigurationseinheit aus. Das bedeutet insbesondere, dass kein *single-point-of-failure* existiert. Zudem handelt es sich bei einer Konfigurationseinheit meist um eine komplexe, schwergewichtige Softwareeinheit, welche insbesondere in mobilen verteilten Anwendungen nicht immer verfügbar und wünschenswert ist.

Mit Hilfe des Einsatzes selbstorganisierender Komponenten lassen sich zudem zahlreiche Anforderungen dynamisch-adaptiver Anwendungen realisieren. Zum einen ist die Integration neuer Komponenten in ein laufendes System möglich, ohne das Gesamtsystem zu kennen. Die neue Komponente muss lediglich installiert werden. Anschließend löst sie selbständig Abhängigkeiten zu anderen Komponenten auf und startet, sobald dies gelungen ist. Das Entfernen einer Komponente aus einer laufenden Anwendung geschieht auf ähnliche Weise. Die Komponente muss lediglich deinstalliert werden und alle Verwender müssen über den Wegfall der Komponente informiert werden. Alle betroffenen Komponenten reagieren dann selbständig auf den Wegfall dieser Komponente.

Mit Hilfe von selbstorganisierenden Komponenten kann eine Anwendung im Grunde selbständig wachsen und sich selbständig rekonfigurieren, indem jede Komponente versucht, ihre lokalen Anforderungen umzusetzen. Man spricht hierbei auch von emergenten Systemeigenschaften.

Der Ansatz und dessen Vorteile lassen sich anhand des Biathlonbeispiels gut erläutern. Es sei hierzu zunächst angenommen, dass lediglich eine Athletenkomponente im System existiert. Diese Komponente bietet einen Dienst an, welcher die Schnittstelle *IAthlete* implementiert und Zugriff auf einen Dienst benötigt, welcher die Schnittstelle *IPulse* implementiert. Weiterhin sei angenommen, dass es sich hierbei um eine autonome, selbstorganisierende Komponente handelt. Die Komponente sucht also nun selbständig nach einem Pulsdienst, den sie verwenden kann, um ihrerseits den Athletendienst anbieten zu können. Die Komponente ist in Abbildung 3-7 dargestellt. Die Eigenschaft, dass sich die Komponente autonom selbst konfiguriert, wird durch einen runden Pfeil symbolisiert.

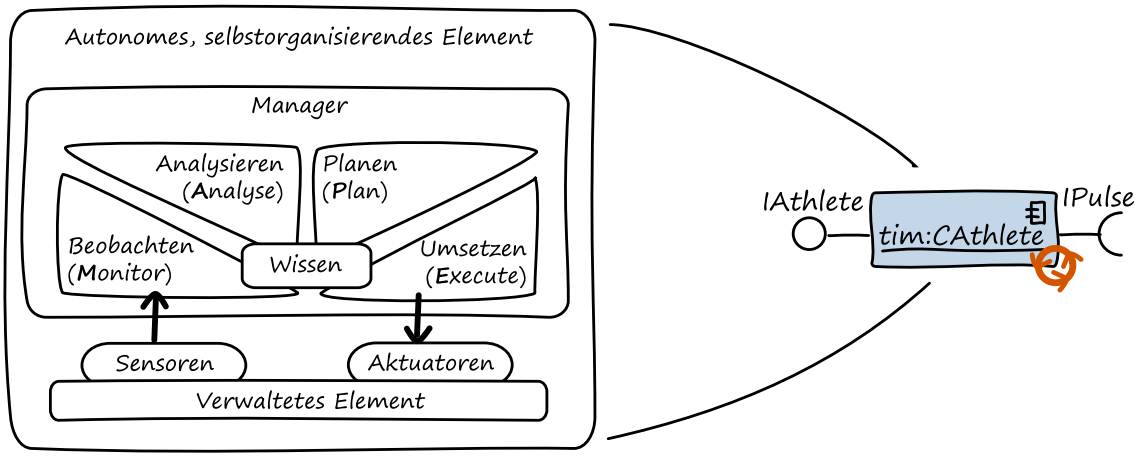


Abbildung 3-7: Eine selbstorganisierende Athletenkomponente

Wird nun eine Pulskomponente dem System hinzugefügt, so verbindet sich die Athletenkomponente mit diesem Dienst und kann ihren eigenen Dienst anbieten. Selbiges passiert, wenn eine Trainerkomponente in das System kommt. Diese wird selbständig nach geeigneten Sportlerkomponenten suchen und sich mit diesen verbinden, um ihrerseits einen Trainerdienst bereitstellen zu können. Das Ergebnis ist in Abbildung 3-8 dargestellt.

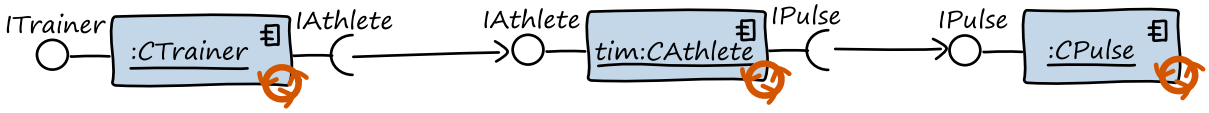


Abbildung 3-8: Ergebnis des Konfigurationsprozesses auf Basis selbstorganisierender Komponenten

Auf diese Weise entsteht im Grunde selbständig, auf Basis autonomer, selbstadaptiver Komponenten, eine Anwendung zur Trainingsunterstützung für Biathleten.

3.2.2 Berücksichtigung Anwendungsarchitektur-spezifischer Anforderungen in selbstorganisierenden Systemen

Eine häufig anzutreffende Anforderung für selbstorganisierende Systeme ist die Notwendigkeit der Einhaltung bestimmter Vorgaben an die Gesamtarchitektur des Systems [MaK96]. Dies kann sowohl strukturelle Aspekte als auch Verhaltensaspekte betreffen. Eine Architektur kann hierbei z.B. Anforderungen hinsichtlich der Existenz bestimmter Komponenten stellen, genauso wie Anforderungen hinsichtlich der Verbindung zwischen Komponenten.

Eine Architekturspezifikation für die Biathlonanwendung könnte beispielsweise fordern, dass genau ein Dienst im System vorhanden sein muss, welcher die Schnittstelle *ISupervisor* implementiert, um ein Schießtraining durchführen zu können. Genauso könnte gefordert sein, dass mindestens zwei Sportlerkomponenten vorhanden sein müssen, welche jeweils mit genau einem Pulssensor verbunden sein sollen. Und schließlich könnte eine weitere Vorgabe lauten, dass die *Supervisor*-Komponente mit genauso vielen Schießbahnkomponenten verbunden sein solle, wie Athleten mit Trainerkomponenten verbunden sind.

Eine solche Architekturspezifikation kann zusätzlich bestimmte Anforderungen hinsichtlich des Zustandes der Dienste definieren. So könnten für eine Biathlonanwendung beispielsweise nur solche Athleten- und Trainerkomponenten zugelassen werden, welche Akteure repräsentieren, die einem bestimmten Team angehören. Die in Abbildung 3-9 dargestellte Konfiguration realisiert beispielsweise die im Absatz zuvor erläuterten Anforderungen.

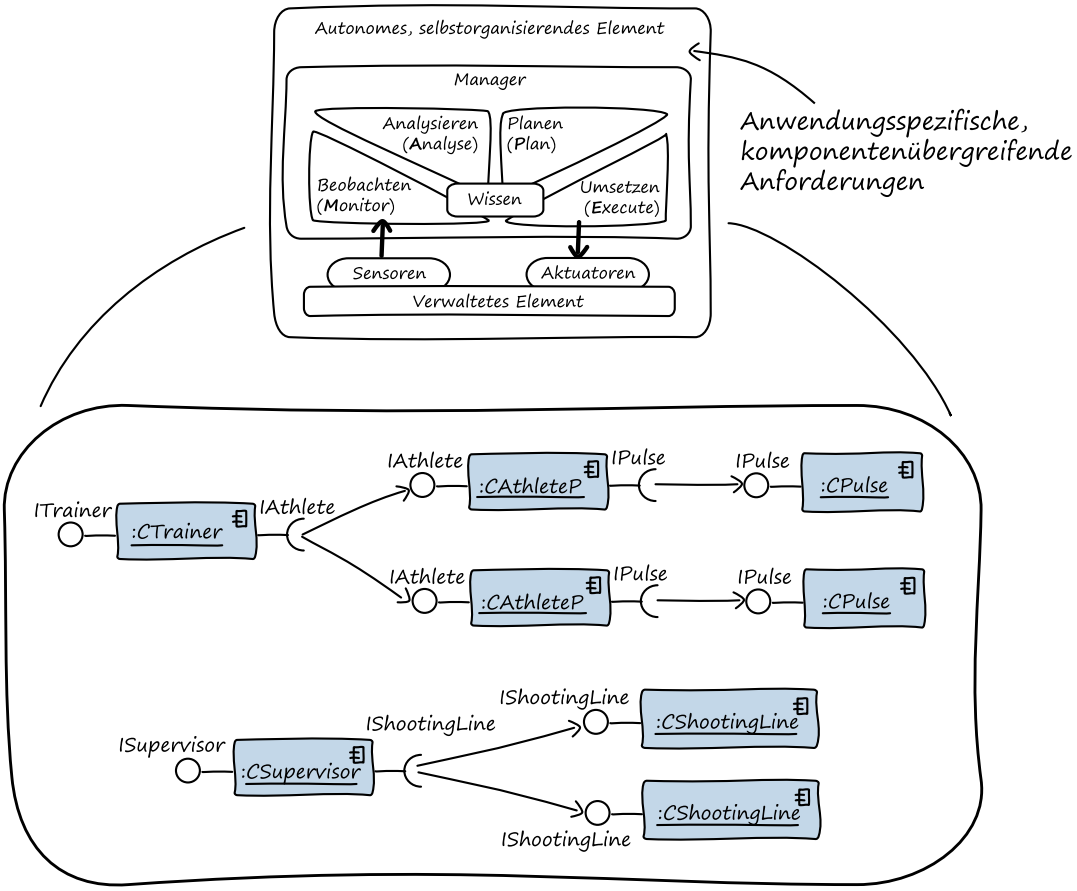


Abbildung 3-9: Eine Anwendungskonfiguration, welche auch globale, anwendungsspezifische Anforderungen berücksichtigt.

Müssen derartige globale Anforderungen berücksichtigt werden, so wird meist auf den Einsatz einer zentralen Konfigurationseinheit zurückgegriffen. Dieser nimmt zum einen die Anforderungen an die zu erzeugende Anwendungskonfiguration entgegen, welche z.B. von einem Anwendungsentwickler bereitgestellt wird. Zum anderen verwaltet dieser sämtliche verfügbare Komponenten und realisiert eine gültige Konfiguration gemäß den hinterlegten Vorgaben. Die Vorteile des Einsatzes selbstorganisierender Komponenten wurden somit aufgegeben auf Kosten der Möglichkeit, anwendungsspezifischen Einfluss auf die Konfiguration zu nehmen.

Aus diesen zwei zunächst gegensätzlichen Ansätzen ergibt sich nun die konkrete Problemstellung, die dieser Arbeit zugrunde liegt. Diese wird im folgenden Abschnitt zusammengefasst.

3.2.3 Problemstellung in der Zusammenfassung

Die Problemstellung, der diese Arbeit zugrunde liegt, ergibt sich aus den zuvor vorgestellten Zielen. Zum einen möchte man eine verteilte mobile Anwendung aus unterschiedlichsten Gründen möglichst aus autonomen, selbstorganisierenden Komponenten zusammensetzen und so ein emergentes Systemverhalten realisieren. Auf der anderen Seite möchte man gleichzeitig die Berücksichtigung anwendungsglobaler Anforderungen sicherstellen. Hierbei sollen sowohl Anforderung an die Struktur einer Anwendung spezifiziert werden können, als auch Anforderungen hinsichtlich des Zustandes angebotener Dienste. Außerdem muss das System in der Lage sein, sowohl die Anforderungen der einzelnen Komponenten automatisiert zu erfüllen, als auch die anwendungsglobalen Anforderungen.

Wird eine Anwendung basierend auf autonomen Komponenten realisiert, ist mit den bislang bekannten Konzepten keine Sicherstellung der Berücksichtigung anwendungsglobaler Anforderungen möglich. Die Alternative hierzu ist die zentrale Steuerung der Konfiguration durch eine Konfigurationseinheit, welche Anforderungen liest, verarbeitet und umsetzt. Alle bisher bekannten Ansätze basieren hierbei darauf, dass die Konfigurationseinheit die Kontrolle über die Komponenten besitzt und zentral entscheidet, welche Komponenten für eine Anwendung verwendet werden, und wie sie vernetzt werden sollen. Außerdem wird in den bisherigen Ansätzen vorausgesetzt, dass innerhalb der Konfigurationsbeschreibung die Komponententypen bereits bekannt sind. Eine nachträgliche Integration neuer Komponenten ist somit nur möglich, indem die Konfigurationsbeschreibung angepasst wird (siehe Abschnitt 3.3).

Die Problemstellung lässt sich somit auf folgende Frage reduzieren:

> *Wie kann eine dynamisch-adaptive Anwendung auf Basis selbstorganisierender Komponenten realisiert werden, welche gleichzeitig Anwendungsarchitektur-spezifische Anforderungen berücksichtigt?*

In den folgenden Kapiteln wird eine Infrastruktur vorgestellt, welche einen Trade-Off darstellt zwischen der Realisierung dynamisch-adaptiver Anwendungen auf Basis selbstorganisierender Komponenten sowie der Möglichkeit, anwendungsspezifisch Einfluss auf die resultierenden Konfigurationen nehmen zu können. Hierbei werden autonome Komponenten zu semi-autonomen Komponenten umgestaltet, indem sie die Möglichkeit anbieten, anwendungsspezifische Anforderungen durch eine leichtgewichtige, zentrale Konfigurationseinheit zu induzieren. Diese zentrale Einheit besitzt hierbei nicht die Kontrolle über die Komponenten, sondern stellt diesen lediglich Informationen bzgl. einzuhaltender Anforderungen für eine Anwendung bereit.

Es wurden bereits einige Arbeiten zur beschriebenen Problemstellung veröffentlicht. Diese werden nun im folgenden Abschnitt vorgestellt.

3.3 Verwandte Arbeiten

Der in dieser Arbeit entwickelte Lösungsansatz hat zum Ziel, die Vorteile von sich autonom konfigurierender Komponenten weitestgehend zu bewahren, bei gleichzeitiger Möglichkeit, komponentenübergreifende Anforderungen spezifizieren und automatisiert umsetzen zu lassen. Die Grundlagen hierfür wurden bereits in Kapitel 2 ausführlich beschrieben, sowie einige Arbeiten zu diesem Thema bereits erwähnt. Einige ausgewählte Arbeiten, welche sich mit dem Thema der Anwendungsarchitektur-konformen Konfiguration selbstorganisierender Komponenten beschäftigen, werden im Folgenden noch einmal kurz diskutiert und mit dem Ansatz der hier vorliegenden Dissertation in Beziehung gesetzt.

Einer der prominentesten Ansätze zur architekturkonformen Selbstadaption ist das Framework *Rainbow* [GCH+04, Che08]. In diesem Ansatz wird die architekturbasierte Selbstadaption als Lösung für den Betrieb und die Wartung komplexer Systeme motiviert. *Rainbow* verwendet ein abstraktes Architekturmodell, um die Ausführung selbstadaptiver Anwendungen zu überwachen und das System ggf. anzupassen. Eine solche Architektur wird repräsentiert als Graph interagierender Elemente, wobei die Knoten Komponenten repräsentieren und Kanten Verbindungen zwischen Komponenten. Beide Elementarten können zusätzlich mit Eigenschaften annotiert werden, wie z.B. der geforderte Durchsatz, Latenz oder Kommunikationsprotokolle. Zur Einhaltung der Anforderungen können Adaptionsstrategien in Form von Methoden hinterlegt werden, in denen die erforderlichen Anpassungsmaßnahmen durchgeführt werden. Zur Entwicklungszeit müssen hierbei die Komponententypen bekannt sein. Instanzen werden hingegen automatisch durch das Framework mittels einer *ResourceDiscovery*-Einheit gefunden und können so dynamisch in laufende Anwendungen eingefügt werden. Später wurde der Ansatz dahingehend erweitert, dass Nutzererfahrungen in den Konfigurationsprozess einfließen können, um bei gegensätzlichen Konfigurationszielen eine geeignete Entscheidung treffen zu können [CGS06]. Es handelt sich bei diesem Ansatz um eine imperative Realisierung der Adaption, da konkrete Adaptionsschritte vom Entwickler programmiert werden müssen. Hierbei kann der Programmieraufwand jedoch schnell sehr umfangreich und komplex werden, da ggf. eine große Zahl von Regeln spezifiziert werden muss und gleichzeitig sich widersprechende Regeln vermieden werden müssen. Die Lösung, welche im Rahmen der Dissertation vorgestellt wird, verfolgt aus diesem Grunde einen deklarativen Ansatz. Hierbei muss der Anwendungsentwickler lediglich die Anwendungsarchitektur-spezifischen Anforderungen definieren, während eine Infrastruktur entscheidet, wie diese Anforderungen auf Basis der zur Verfügung stehenden Komponenten umgesetzt werden können. Ein weiterer Nachteil des *Rainbow*-Ansatzes ist, dass die Komponententypen zur Entwicklungszeit der Anwendung bekannt sein müssen. Dies hat zur Folge, dass die Integration neuer Komponenten nur durch Anpassung des Architekturmodells erfolgen kann. Die hier vorliegende Dissertation ermöglicht hingegen die Definition eines Architekturmodells, ohne dass Komponententypen bekannt sein müssen. Hierdurch wird somit der emergente Charakter mobiler verteilter Anwendungen berücksichtigt.

Einen ähnlichen Nachteil besitzt das Framework, welches in [RaP03] vorgestellt wurde. Es handelt sich hierbei um ein .NET-basiertes Framework, welches die Umsetzung komponentenübergreifender Anforderungen ermöglicht. Jedoch müssen auch hier alle beteiligten Komponenten zur Spezifikationszeit des Systems bekannt sein. Vielmehr ist es hier so, dass nicht nur die Komponententypen, sondern auch die konkreten Instanzen bekannt sein

müssen. Die Lösung basiert darauf, dass ein Anwendungsentwickler die Menge aller valider Konfigurationen definiert. Zur Laufzeit wird dann automatisiert diejenige Konfiguration geladen, welche zur aktuellen Situation am besten passt. Zu diesem Zweck hinterlegt Anwendungsentwickler entsprechende Kriterien zur Aktivierung einer Konfiguration im Framework. Doch gerade in mobilen verteilten Systemen ist es nur bedingt möglich, für jede möglicherweise auftretende Situation eine entsprechende Konfiguration vorab zu definieren [CCS09]. Aus diesem Grunde wird in der vorliegenden Dissertation eine Lösung vorgeschlagen, die auf der Spezifikation einer abstrakten Konfiguration durch den Anwendungsentwickler beruht. Innerhalb dieser abstrakten Konfiguration entscheidet das Framework zur Laufzeit autonom, welche Konfiguration die Anforderungen dieser abstrakten Konfiguration erfüllt und erzeugt diese.

In [GMK02] wurde, wie in dieser Arbeit auch, das Problem bearbeitet, aus einer Menge autonomer Komponenten ein System zu erzeugen, welches komponentenübergreifende Anforderungen erfüllt. Hierbei beruht der Ansatz darauf, dass jede Komponente eine vollständige Sicht auf das System besitzt. Auf Basis dieser vollständigen Laufzeitsicht können für eine Komponente anwendungsspezifische, komponentenübergreifende Bedingungen spezifiziert werden, die diese Komponente bei der Auflösung von Abhängigkeiten zu anderen Komponenten berücksichtigt. Zu diesen Bedingungen müssen zusätzlich die konkreten Konfigurationsschritte definiert werden, die im Falle einer auftretenden Verletzung der Bedingungen ausgeführt werden. Komponentenübergreifende Anforderungen können insofern berücksichtigt werden, als dass sowohl die Bedingungen als auch die Konfigurationsschritte Zugriff auf die vollständige Systemsicht haben und hierauf Bezug nehmen können. Ein wesentlicher Nachteil dieses Ansatzes ist die getroffene Annahme, dass jede Komponente eine vollständige Sicht auf das System besitzt. Hierzu ist ein hoher Kommunikationsaufwand erforderlich, um jede Veränderung allen Komponenten mitzuteilen, so dass diese ihre interne Repräsentation des Gesamtsystems aktuell halten können. Und genau wie im *Rainbow*-Framework handelt es sich hier um einen imperativen Ansatz, mit den bereits diskutierten Nachteilen. Die Umsetzung orientiert sich, genau wie die hier vorgestellte Lösung auch, an das bereits in Abschnitt 2.3 beschriebene Referenzmodell für selbstorganisierende Systeme.

In [LGS+06] wurde ein Framework namens *dAcme* für verteilte, selbstadaptive Systeme vorgestellt. Auch dieses Framework hat zum Ziel, die Einhaltung Anwendungsarchitektur-spezifischer Anforderungen zur Laufzeit zu gewährleisten. Hierzu sieht der Ansatz vor, zunächst eine initiale Konfiguration auf Instanzebene zu beschreiben und auszuführen. Die Beschreibung erfolgt hierbei mit Hilfe der Architekturbeschreibungssprache *Acme* [GMW00]. Für eine Anwendung können nun Invarianten hinterlegt werden. Zu jeder Invariante kann ein Konfigurationsprogramm (in *dAcme* auch Taktik genannt) hinterlegt werden, welches automatisch ausgeführt wird, sobald die Bedingung der Invariante verletzt ist. Auch dieser Ansatz beruht wieder auf der Annahme, dass Komponententypen und Instanzen zur Entwicklungszeit der Anwendung bekannt sind. Zudem handelt es sich hierbei um ein Verfahren, das die Existenz einer zentralen Konfigurationseinheit voraussetzt, welche eine vollständige Sicht auf das System benötigt. Im Gegensatz dazu bietet die Lösung, welche in dieser Dissertation im Folgenden vorgestellt wird, die Möglichkeit, auf eine zentrale Konfigurationseinheit zu verzichten, falls keinerlei komponentenübergreifende Anforderungen berücksichtigt werden müssen. Besteht hingegen die Notwendigkeit der Berücksichtigung Anwendungsarchitektur-spezifischer Anforderungen, so muss zwar eine zentrale

Konfigurationseinheit eingeführt werden, diese benötigt allerdings keine vollständige Sicht auf das Gesamtsystem inklusive aller existierenden Verbindungen zwischen Komponenten.

Alle bislang vorgestellten Arbeiten setzen voraus, dass die Komponententypen zur Entwicklungszeit der Anwendung bekannt sind. In [CPC08] wird hingegen ein Ansatz vorgestellt, der die Integration von Komponententypen, die zum Entwicklungszeitpunkt der Anwendung noch nicht bekannt waren, ermöglicht. Hierbei wird insbesondere die Integration neuer Komponenten in laufende Anwendungen betrachtet. Die Anwendungsarchitektur wird bei diesem Ansatz mit Hilfe einer aspektorientierten Architekturbeschreibungssprache [PAC06] beschrieben. Diese beinhaltet im Wesentlichen die Definition von Komponententypen sowie Verbindungen zwischen diesen. Zusätzlich können u.a. Anforderungen hinsichtlich der Anzahl zu erzeugender Komponenteninstanzen je Komponententyp festgelegt werden, sowie hinsichtlich der Verbindungen zwischen diesen Instanzen. Zum Start einer Anwendung werden die Komponententypen gemäß der Spezifikation initial instanziiert und die Verbindungen zwischen diesen entsprechend hergestellt. Zur Laufzeit konfiguriert sich das System selbständig, so dass stets die Vorgaben, welche durch die Anwendungsarchitektur gegeben sind, eingehalten werden. Die Realisierung der Anpassbarkeit orientiert sich wiederum am bereits in Abschnitt 2.3 vorgestellten Referenzmodell für selbstorganisierende Systeme. Die Besonderheit des Ansatzes ist nun, dass sich die zugrundeliegende Anwendungsarchitektur ebenfalls zur Laufzeit anpassen lässt. Allerdings müssen diese Änderungen durch beispielsweise einen Administrator in das System eingepflegt werden. In der hier vorliegenden Dissertation ist zwar nicht vorgesehen, dass die Anwendungsarchitektur zur Laufzeit verändert wird. Jedoch ist es auch hier möglich, neue Komponententypen zur Laufzeit in die Anwendung zu integrieren. Im Gegensatz zum soeben vorgestellten Ansatz ist hierfür kein manueller Eingriff in das System notwendig. Vielmehr können diese Komponenten automatisch an geeigneter Stelle in die Anwendung integriert werden.

Eine Lösung zur Realisierung emergenten Systemverhaltens durch den Einsatz selbstorganisierender Komponenten wurde in [BNM+08] vorgestellt. Selbstorganisierende Einheiten werden hier als *SelfLets* bezeichnet. Diese sind autonome Einheiten, welche in der Lage sind, automatisiert Abhängigkeiten zu anderen *SelfLets* zur Laufzeit aufzulösen, Änderungen in der Ausführungsumgebung zu erfassen und anhand von Richtlinien (*Policies*) angemessen auf derartige Änderungen zu reagieren. Genau wie in einigen bereits zuvor vorgestellten Lösungen orientiert sich auch der *SelfLet*-Ansatz am Referenzmodell für selbstorganisierende Systeme (siehe Abschnitt 2.3), wobei hier jede Komponente als eigenständiges selbstorganisierendes System betrachtet wird. Der *SelfLet*-Ansatz bietet allerdings nicht die Möglichkeit, Anwendungsarchitektur-spezifische Anforderungen definieren und umsetzen zu können. Allerdings wäre eine Kombination des *SelfLets*-Ansatzes mit der hier vorgestellten Lösung zur Anwendungsarchitektur-konformen Konfiguration durchaus denkbar.

In [Ald03] wird insbesondere das Problem der Inkonsistenz zwischen Anwendungsarchitektur-beschreibung und zugehöriger Implementierung behandelt. Hier wird argumentiert, dass insbesondere die Evolution eines Softwaresystems dazu führt, dass derartige Inkonsistenzen entstehen. Der vorgestellte Lösungsansatz basiert darauf, die jeweils verwendete Programmiersprache derart zu erweitern, dass Architekturregeln in der Implementierung hinterlegt und automatisch zur Laufzeit überprüft werden können. Der Ansatz wurde in Form der Sprache *ArchJava* umgesetzt. Auf diese Weise lassen sich Bedingungen an die Struktur einer

Anwendung definieren, allerdings ist der Ansatz weder auf verteilte Anwendungen ausgelegt, noch auf das dynamische Hinzukommen und Wegfallen einzelner Komponenten.

Dieser Abschnitt hatte lediglich zum Ziel, einen groben Überblick über einige existierende Forschungsarbeiten auf dem Gebiet der Anwendungsarchitektur-konformen Konfiguration in selbstorganisierenden Systemen zu geben, um die im Folgenden vorgestellte Lösung einordnen zu können. In [DJM+04] und anderen Arbeiten werden weitere Lösungsansätze umrissen, sowie die Relevanz des Themas herausgestellt. Zusammenfassend lässt sich sagen, dass zwar bereits einige Arbeiten zur Einhaltung Anwendungsarchitektur-spezifischer Anforderungen in selbstorganisierenden Systemen existieren, diese jedoch meist darauf basieren, dass Komponententypen und Instanzen zur Entwicklungszeit der Anwendung bekannt sein müssen. Gerade in emergenten Systemen kann hiervon nicht immer ausgegangen werden. Außerdem existieren einige Ansätze zur Realisierung eines emergenten Systemverhaltens, meist basierend auf autonomen, selbstorganisierenden Komponenten. Hierbei werden allerdings nur sehr bedingt Anwendungsarchitektur-spezifische Anforderungen während des Konfigurationsprozesses berücksichtigt.

In den folgenden zwei Kapiteln wird nun zunächst ein Komponentenmodell vorgestellt, welches die Erstellung und Ausführung von emergenten Systemen basierend auf selbstorganisierenden Komponenten ermöglicht, mit dem Ziel, für jede Komponente lokal die beste Konfiguration zu erzeugen. In Kapitel 6 wird basierend auf dem Komponentenmodell dann die Lösung dahingehend erweitert, dass Anwendungsarchitektur-spezifische Anforderungen definiert werden können. Der Ansatz erlaubt zudem die automatisierte Erzeugung einer Anwendungsarchitektur-konformen Konfiguration, ohne dass konkrete Komponententypen und Instanzen zur Entwicklungszeit der Anwendung bekannt sein müssen. Hierdurch kann der emergente Charakter des Systems weitestgehend erhalten werden.

4 Eine Infrastruktur für selbstorganisierende Softwaresysteme

In diesem Kapitel wird eine Infrastruktur zur Entwicklung und Ausführung selbstorganisierender Softwaresysteme vorgestellt. Diese basiert zunächst ausschließlich auf selbstorganisierenden Komponenten und kommt ohne den Einsatz eines zentralen Konfigurators aus. Mit Hilfe der Infrastruktur lassen sich somit Systeme mit emergenten Eigenschaften erzeugen. Kern der Infrastruktur ist ein Framework für selbstorganisierende Komponenten. Dieses Framework übernimmt sämtliche Aufgaben, die die Realisierung selbstorganisierender Eigenschaften betreffen, so dass sich Komponentenentwickler ganz auf die Implementierung der fachlichen Funktionalität fokussieren können. Die Eigenschaften und verfolgten Ziele werden im folgenden Abschnitt kurz zusammengefasst, bevor anschließend die erarbeitete Lösung im Detail vorgestellt wird.

4.1 Ziele und Eigenschaften der Infrastruktur

Wie bereits angedeutet wurde, wird in diesem Kapitel eine Infrastruktur vorgestellt, die die Entwicklung und Ausführung von Systemen mit emergenten Eigenschaften unterstützt. Hierbei wird auf den Einsatz eines zentralen Konfigurators verzichtet, um so von den Vorteilen eines dezentral konfigurierenden Systems profitieren zu können. Die Grundidee hierbei ist, dass jede Komponente die MAPE-K-Schleife realisiert (siehe Abschnitt 2.3), wie in Abbildung 4-1 dargestellt.

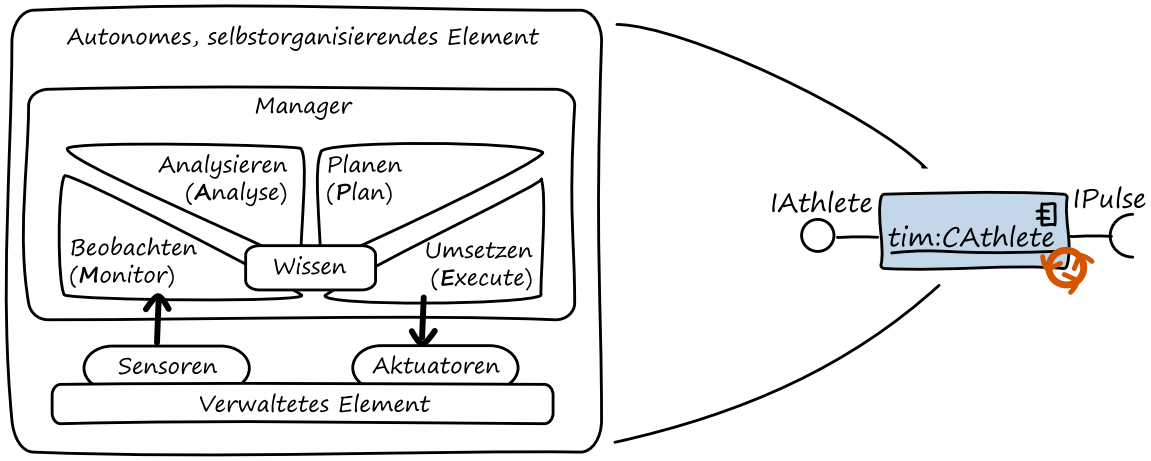


Abbildung 4-1: Jede Komponente realisiert die MAPE-K-Schleife

Jede Komponente besitzt somit Wissen darüber, welche lokalen Ziele bei der Rekonfiguration berücksichtigt werden müssen und reagiert autonom auf Veränderungen in der Umgebung. Auf welche Ereignisse das hier vorgestellte Framework genau in der Lage ist, automatisiert zu reagieren, wird in den folgenden Abschnitten erläutert. Diese Fähigkeiten werden in den folgenden zwei Kapiteln der Arbeit dann schrittweise erweitert.

4.1.1 Automatisches Auflösen von Abhängigkeiten

Häufig benötigen Komponenten Zugriff auf Funktionalität anderer Komponenten, um ihrerseits Dienste anbieten zu können. Das Framework ist hierbei in der Lage, diese Abhängigkeiten aufzulösen und hierbei komponentenspezifische Anforderungen zu berücksichtigen. Diese

Anforderungen werden durch den Komponentenentwickler vorgegeben. Das Framework wertet diese Anforderungen zur Laufzeit aus und berücksichtigt sie bei der Auflösung der Abhängigkeiten. Zum einen kann spezifiziert werden, für welche Schnittstellen die Komponente eine Implementierung durch andere Komponenten benötigt. Für die in Abbildung 4-2 dargestellte Athletenkomponente wurde beispielsweise eine Abhängigkeit zu einer Komponente definiert, die die Schnittstelle *IPulse* implementiert.

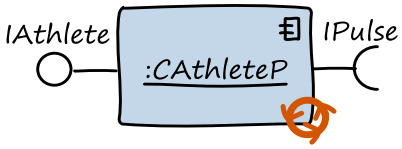


Abbildung 4-2: Komponente mit einer Abhängigkeit zu einem Pulssensor

Das Framework wird zur Laufzeit entsprechend nach Komponenten suchen, die diese Schnittstelle implementieren. Sobald eine solche Komponente gefunden wurde, stellt das Framework dem fachlichen Teil der Komponente die Referenz hierauf zur Verfügung. Für viele Anwendungen reicht diese Möglichkeit allerdings nicht aus. So ist es beispielsweise für die Athletenkomponente nicht sinnvoll, dass sie zwei oder mehr Pulssensoren gleichzeitig nutzt. Das Framework sieht deshalb die Möglichkeit vor, die Anzahl benötigter Referenzen einzuschränken bzw. vorzugeben. So lässt sich für die benötigte Schnittstelle der Pulskomponente die Anzahl auf Eins beschränken, wie in Abbildung 4-3 gezeigt. Auf der anderen Seite können angebotene Dienste in einigen Fällen nicht von beliebig vielen Nutzern gleichzeitig verwendet werden. Die maximale Anzahl von Dienstnutzern wird in der hier verwendeten Notation an den angebotenen Dienst notiert, wie in Abbildung 4-3 dargestellt. Ein Stern symbolisiert, dass der Dienst der Dienst von beliebig vielen Dienstnutzern gleichzeitig verwendet werden darf.

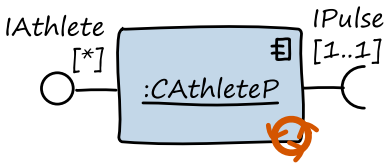


Abbildung 4-3: Angabe von oberen und unteren Schranken

In diesem Fall würde das Framework dafür sorgen, dass der angebotene Dienst der Athletenkomponente erst dann ausgeführt wird, wenn die Komponente mit genau einem Pulsdienst verbunden ist. Stehen mehr als ein Dienst im System bereit, so wählt das Framework einen beliebigen aus. Umfangreichere Möglichkeiten hierzu bietet die Erweiterung, welche in Kapitel 5 vorgestellt werden.

Jede Komponente bietet einen oder mehrere Dienste an. Diese können von anderen Komponenten verwendet werden, um ihrerseits Dienste anzubieten. In zahlreichen Situationen sind Dienste allerdings nicht auf die gleichzeitige Nutzung durch beliebig viele Nutzer ausgelegt. Aus diesem Grunde bietet das im weiteren Verlauf des Kapitels vorgestellte Framework die Möglichkeit, eine maximale Anzahl von Nutzern festzulegen. Das Framework sorgt zur Laufzeit dafür, dass diese Anzahl nicht überschritten wird.

4.1.2 Automatische Reaktion auf Ereignisse

Gerade in mobilen verteilten Softwaresystemen kann es häufig vorkommen, dass die Verbindung zwischen Komponenten unterbrochen wird oder neue Komponenten hinzukommen und gestartet werden. Jede Komponente muss hierbei in der Lage sein, hierauf angemessen zu reagieren. Das hier vorgestellte Framework ist in der Lage, sowohl auf das Hinzukommen als auch auf den Wegfall von Komponenten autonom zu reagieren. Die durch das Framework realisierten Reaktionen werden nun kurz beschrieben.

Benötigt eine Komponente Zugriff auf die Funktionalität anderer Komponenten (definiert sie also benötigte Schnittstellen), so können die angebotenen Dienste der Komponente erst dann ausgeführt werden, wenn alle Abhängigkeiten aufgelöst werden konnten. Für die Athletenkomponente aus dem vorherigen Abschnitt bedeutet das z.B., dass die Schnittstelle *IAthlete* erst dann von anderen Komponenten verwendet werden kann, wenn die Komponente Zugriff auf einen Pulssensor hat. Ist also keine Pulskomponente verfügbar, kann die Athletenkomponente und dessen angebotener Dienst nicht ausgeführt werden. Kommt nun eine Pulskomponente in das System, so reagiert das Framework, indem es versucht, Zugriff auf den Dienst zu bekommen. Ist dies erfolgreich, so wird die Athletenkomponente, bzw. dessen angebotener Dienst, durch das Framework gestartet. Fällt die Pulskomponente hingegen aus, so wird auch dies durch das Framework erkannt. Das Framework versucht dann zunächst, eine Ersatzkomponente im System zu nutzen. Steht keine Pulskomponente zur Verfügung, so beendet das Framework die Ausführung der Athletenkomponente.

In vielen Fällen definiert eine Komponente mehrere angebotene und/oder benötigte Schnittstellen. In Abbildung 4-4 ist beispielsweise eine Athletenkomponente dargestellt, die zwei Dienste anbietet und einen benötigt.

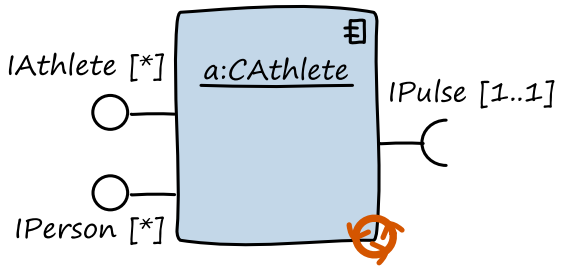


Abbildung 4-4: Athletenkomponente mit zwei angebotenen und einer benötigten Schnittstelle

Über die Schnittstelle *IPerson* können Informationen wie Name oder Team des Athleten abgefragt werden. Über die Schnittstelle *IAthlete* kann hingegen die aktuelle Herzfrequenz abgerufen werden. Beide Dienste können mit den bislang skizzierten Konzepten erst dann ausgeführt werden, wenn ein Pulssensor verfügbar ist. Jedoch wird zur Realisierung der Schnittstelle *IPerson* eigentlich kein Pulssensor benötigt. Für derartige Fälle sieht die im Folgenden vorgestellte Lösung ein besonderes Konzept vor. Das Framework ermöglicht Komponentenentwicklern, innerhalb einer Komponente verschiedene Abbildungen von benötigten auf angebotene Dienste zu spezifizieren. Eine solche Abbildung wird im Folgenden als *Komponentenkonfiguration* bezeichnet, wobei zu jedem Zeitpunkt maximal eine Komponentenkonfiguration aktiv ist. Auf diese Weise können Komponenten ihre Dienste nicht nur ganz oder gar nicht anbieten, sondern je nach aufgelösten Abhängigkeiten auch ganz bestimmte Dienste. Das Konzept wird im Laufe des Kapitels noch im Detail vorgestellt. Das

Framework sorgt zur Laufzeit dafür, dass die bestmögliche *Komponentenkonfiguration* aktiviert und die entsprechenden Dienste gestartet werden.

4.1.3 Steuerung des Lebenszyklus

Das Framework übernimmt die Kontrolle über den gesamten Lebenszyklus der Komponente. Es sorgt dafür, dass Abhängigkeiten gemäß komponentenspezifischer Anforderungen aufgelöst werden, dass Komponentenkonfigurationen aktiviert/deaktiviert und entsprechende Dienste aktiviert/deaktiviert werden. Als Komponentenentwickler kann man sich somit nahezu ausschließlich auf die Realisierung fachlicher Anforderungen konzentrieren. Der fachliche Teil der Komponente wird durch das Framework lediglich benachrichtigt, falls die Ausführung bestimmter Dienste gestartet oder gestoppt werden muss. Außerdem werden ihr Referenzen auf Dienste anderer Komponenten bereitgestellt oder entzogen.

Mit diesem Konfigurationsprozess beginnt das Framework allerdings erst unter bestimmten Voraussetzungen. In der hier präsentierten Lösung werden Dienste nämlich erst dann gestartet, wenn hierfür auch potentielle Nutzer vorhanden sind. Auf diese Weise können insbesondere Ressourcen gespart werden. Nutzer eines Dienstes können entweder andere Komponenten sein, oder Endanwender. So wird ein Pulsdienst beispielsweise erst dann gestartet werden, wenn ein Nutzer in Form einer anderen Komponente vorliegt (z.B. in Form einer Athletenkomponente). Wie später noch ausführlicher erläutert wird, melden Komponenten hierzu einen entsprechenden Nutzungsbedarf beim Dienst an. Dienste, die hingegen vorrangig für Endanwender interessant sind, müssen durch den Komponentenentwickler explizit als solche gekennzeichnet werden. Für diese Dienste wird das Framework versuchen, die definierten Abhängigkeiten auch dann aufzulösen um sie auszuführen, auch wenn kein Nutzer in Form anderer Komponenten vorhanden ist.

In den folgenden Abschnitten wird nun erläutert, wie die soeben vorgestellten Ziele und Funktionalitäten umgesetzt wurden. Grundlage der Lösung ist hierbei ein Komponentenmodell. Dieses beschreibt zum einen, aus welchen Elementen eine Komponente zusammengesetzt werden kann, und zum anderen, wie die Fähigkeit zur Selbstorganisation umgesetzt wurde.

4.2 Überblick über das Komponentenmodell für selbstorganisierende Komponenten

Wie bereits zuvor erläutert, wird in diesem Kapitel ein Framework vorgestellt, welches die Entwicklung selbstorganisierender Softwaresysteme unterstützt. Komponenten müssen hierzu konform zu einem Komponentenmodell entwickelt werden, welches vom Framework definiert wird. In diesem Abschnitt wird das Komponentenmodell und seine Bestandteile im Detail vorgestellt. Dieses Modell dient im weiteren Verlauf als Basis für eine Infrastruktur, welche die architekturkonforme Konfiguration selbstorganisierender Softwaresysteme ermöglicht.

Das entwickelte Komponentenmodell, welches dem Framework zugrunde liegt, ist in Abbildung 4-5 dargestellt. Es handelt sich hierbei um eine Darstellung als UML Klassendiagramm [Obj09]. Jeder dieser Klassen repräsentiert einen Bestandteil einer selbstorganisierenden Komponente. Die eingezeichneten Assoziationen spiegeln entsprechend die Beziehung zwischen diesen Elementen wieder.

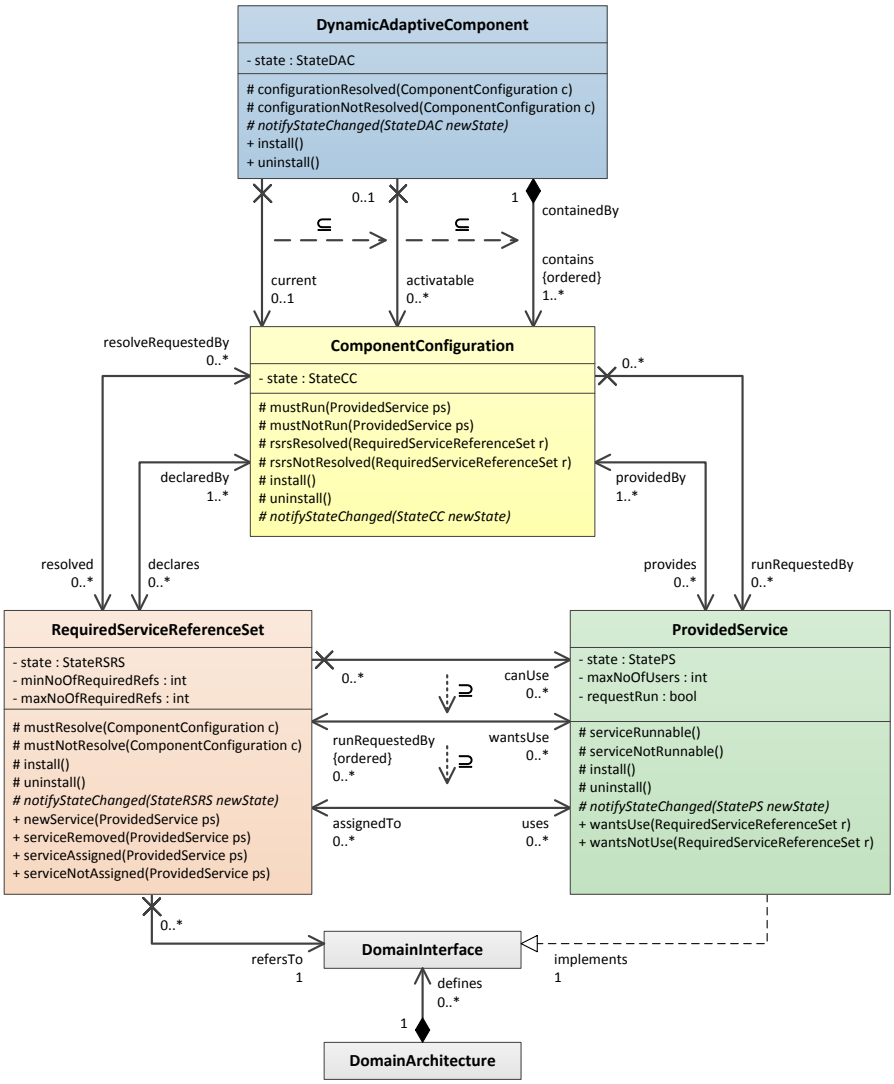


Abbildung 4-5: Komponentenmodell zur Realisierung selbstkonfigurierender Komponenten

Das Komponentenmodell definiert sechs Klassen: *DynamicAdaptiveComponent, ComponentConfiguration, RequiredServiceReferenceSet, ProvidedService, DomainInterface* sowie *DomainArchitecture*. Hierbei definieren die vier erstgenannten Klassen die Bestandteile einer Komponente. Im Folgenden werden die einzelnen Elemente und deren Zusammenspiel kurz umrissen, bevor dann in den folgenden Abschnitten eine detaillierte Erläuterung folgt.

Jede Instanz der Klasse *DynamicAdaptiveComponent* repräsentiert genau eine Instanz einer selbstorganisierenden Komponente. Komponenten dienen dazu, bestimmte Funktionalität zu realisieren und anderen Komponenten oder Endanwendern zur Verfügung zu stellen. Im vorliegenden Modell stellt eine Komponente ihre Funktionalität über angebotene Dienste zur Verfügung. Jeder angebotene Dienst wird im Modell durch eine Instanz der Klasse *ProvidedService* repräsentiert.

Für potentielle Nutzer von Diensten ist es entscheidend zu wissen, welche Funktionalität durch einen angebotenen Dienst einer Komponente erbracht wird. Hierbei werden je nach Anwendungsdomäne unterschiedliche Funktionalitäten benötigt. So sind innerhalb der Biathlondomäne beispielsweise Dienste relevant, welche den aktuellen Puls oder die aktuelle Position und Geschwindigkeit eines Sportlers bereitstellen. Potentielle Dienstnutzer müssen

zudem wissen, wie sie derartige Informationen bei einem Dienst abrufen können. Hierzu sieht das Modell das Konzept einer Domänenarchitektur vor, in der alle für eine Domäne relevanten Schnittstellen definiert sind. Jede Schnittstelle definiert hierbei die Syntax und Semantik beliebig vieler Methoden. So könnte hier beispielsweise eine Schnittstelle *IPulse* hinterlegt sein, mit der Information, dass über diese der aktuelle Puls eines Sportlers erfragt werden kann. Des Weiteren könnte innerhalb der Schnittstelle eine Methode *getPulse():int* definiert sein, über die der aktuelle Puls abgefragt werden kann. Die Domänenschnittstellen für das Biathlonbeispiel sind in Abbildung 3-1 dargestellt. Möchte nun eine Komponente anderen Komponenten den aktuellen Puls eines Sportlers zur Verfügung stellen, so kann er einen Dienst anbieten, welcher die Schnittstelle *IPulse* realisiert. Jeder angebotene Dienst implementiert stets genau eine solche Domänenschnittstelle, wobei eine Komponente beliebig viele Dienste anbieten kann.

Häufig müssen angebotene Dienste einer Komponente auf Dienste anderer Komponenten zugreifen, um die Funktionalität zu realisieren. So kann beispielsweise eine Athletenkomponente den aktuellen Puls nur dann einem Trainer zur Verfügung stellen, wenn sie ihrerseits Zugriff auf eine Pulskomponente besitzt. Derartige Abhängigkeiten zu Diensten anderer Komponenten können im Modell über Instanzen der Klasse *RequiredServiceReferenceSet* definiert werden. Jeder dieser Instanzen repräsentiert hierbei nicht die Abhängigkeit zu genau einem angebotenen Dienst einer anderen Komponente, sondern zu einer Menge von Diensten. Genau wie angebotene Dienste einer Komponente auch, referenziert jedes *RequiredServiceReferenceSet* hierzu genau eine Domänenschnittstelle. Und ein *RequiredServiceReferenceSet* definiert dann Abhängigkeiten zu solchen Diensten, welche die referenzierte Domänenschnittstelle implementieren.

Wie bereits erwähnt, können bestimmte Dienste erst dann ausgeführt werden, wenn sie Zugriff auf Dienste anderer Komponenten haben. Um diese Beziehungen zwischen angebotenen und benötigten Diensten einer Komponente definieren zu können, wurde im Modell die Klasse *ComponentConfigurations* eingeführt. Hierbei handelt es sich um die Realisierung des Konzeptes der *Komponentenkonfigurationen,* welches bereits im vorangegangenen Abschnitt kurz vorgestellt wurde. Jede Komponente definiert mindestens eine solche *ComponentConfiguration*. Immer dann, wenn alle Abhängigkeiten einer Komponentenkonfiguration aufgelöst wurden, können auch die mit ihr verknüpften angebotenen Dienste ausgeführt werden. Wie später in diesem Kapitel noch gezeigt wird, ist zu jedem Zeitpunkt maximal eine *ComponentConfiguration* aktiv.

Die Details zu den einzelnen Klassen des Komponentenmodells sowie deren Zusammenwirken werden in den folgenden Abschnitten dieses Kapitels ausführlich vorgestellt. Im weiteren Verlauf der Arbeit wird für Komponenten eine einheitliche grafische Notation verwendet, welche sich an die in [NKA+07] eingeführte Notation anlehnt. Anhand des in Abbildung 4-6 dargestellten Beispiels werden die einzelnen Notationselemente kurz erläutert.

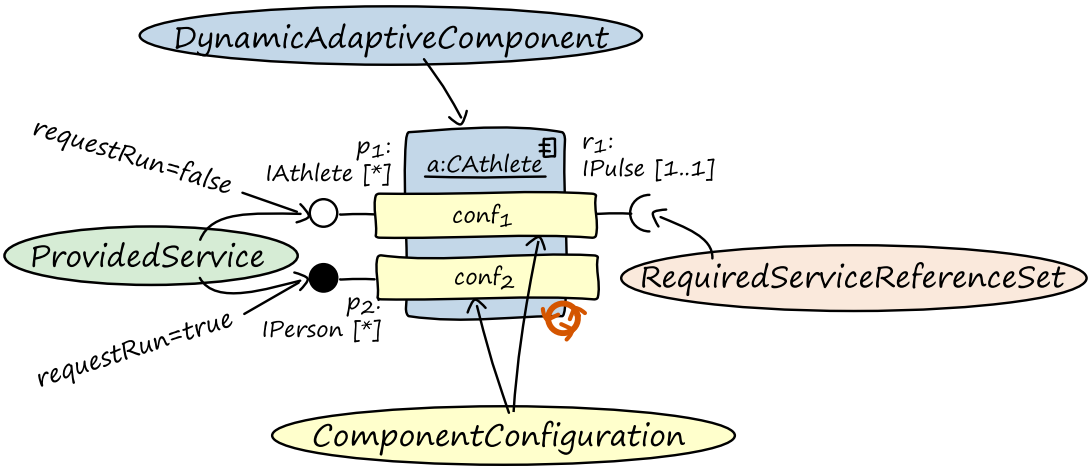


Abbildung 4-6: Beispielnotation einer Komponente

Eine Instanz von *DynamicAdaptiveComponent* wird als Rechteck notiert, wobei oben rechts das aus UML-Komponentendiagrammen bekannte Symbol einer Komponente dargestellt wird. Der Kreis mit den Pfeilen unten rechts symbolisiert, dass es sich um eine selbstorganisierende Komponente handelt. Die Komponente wird zudem beschriftet mit einem eindeutigen Bezeichner der Instanz (in diesem Fall *a*), sowie mit dem Typ der Komponente. Die unterstrichene Schreibweise symbolisiert, dass es sich um die Darstellung einer Instanz handelt. *ComponentConfigurations* werden als Querbalken über dieses Rechteck gelegt. Wie später gezeigt wird, muss über die Komponentenkonfigurationen eine totale Ordnung definiert werde. Die „beste“ *ComponentConfiguration* wird hierbei oben notiert. Angebotene Dienste werden als Kreise notiert und ebenfalls mit einem eindeutigen Bezeichner versehen (in diesem Fall p_1 und p_2). Ein ausgefüllter Kreis symbolisiert einen Dienst, der möglichst auch dann vom Framework ausgeführt werden soll, wenn kein Verwender in Form einer anderen Komponente existiert. Dahinter wird die Domänenschnittstelle angegeben, welche vom jeweiligen Dienst implementiert wird. Außerdem wird die maximale Anzahl von Nutzern eines Dienstes in eckigen Klammern angegeben. Jedes *RequiredServiceReferenceSet* wird mit einem Halbkreis notiert, welcher ebenfalls mit einem eindeutigen Bezeichner versehen wird, sowie mit der referenzierten Domänenschnittstelle. Außerdem wird die minimale und maximale Anzahl benötigter Referenzen in eckigen Klammern notiert. In Abbildung 4-6 benötigt die Komponente beispielsweise genau eine Referenz auf einen Dienst, welcher die Domänenschnittstelle *IPulse* implementiert. Diese Sachverhalte werden allerdings im Verlauf dieses Kapitels noch im Detail vorgestellt.

Zunächst wird nun im nächsten Abschnitt das Konzept der Domänenarchitektur erläutert, sowie deren Zweck. Im Anschluss daran werden die übrigen Klassen vorgestellt. Eine Besonderheit stellen die Assoziationen zwischen den Klassen *RequiredServiceReferenceSet* und *ProvidedService* dar. Sie repräsentieren Beziehungen zwischen Komponenten und werden in einem gesonderten Abschnitt eingeführt.

4.3 Die Domänenarchitektur

Das Domänenmodell bildet die Basis der Kommunikation zwischen Komponenten. Die meisten Komponenten sind auf die Funktionalität anderer Komponenten angewiesen, um ausgeführt werden zu können. Im hier vorgestellten Ansatz stellen Komponenten ihre Funktionalität

dadurch zur Verfügung, dass sie eine oder mehrere Schnittstellen implementieren. Sowohl Anbieter als auch Nutzer müssen sich über Syntax und Semantik dieser Schnittstellen und der darin definierten Methoden einig sein. Ist dies nicht der Fall, so wird das System nicht korrekt funktionieren können.

Aus diesem Grunde liegt jedem System eine sogenannte Domänenarchitektur zugrunde. Eine Domäne ist im Grunde ein bestimmtes Fachgebiet mit allgemein anerkanntem Sprachgebrauch und Begriffsdefinitionen. Innerhalb einer Domänenarchitektur wird dieses Domänenwissen abgelegt und für die Allgemeinheit zugänglich gemacht.

Im Modell aus Abbildung 4-5 ist die Domänenarchitektur durch die Klasse *DomainArchitecture* repräsentiert. Diese definiert eine Menge von Schnittstellen einer Domäne. Diese sogenannten Domänenschnittstellen sind im Modell durch die Klasse *DomainInterface* repräsentiert. Jede Domänenarchitektur kann beliebig viele Domänenschnittstellen definieren. Dieser Sachverhalt ist im Modell durch die Assoziation *defines* repräsentiert. Im Folgenden werden sämtliche Assoziationen durch Relationen formalisiert. Die angegebenen Klassen definieren hierbei die Basismengen. Die Formalisierung für die Assoziation *defines* ist in Formel 4-1 angegeben:

$$\boldsymbol{defines}: DomainArchitecture \rightarrow \mathcal{P}(DomainInterface)$$

Formel 4-1

Jede Domänenschnittstelle kann eine beliebige Anzahl von Methoden definieren. Hierbei wird im Domänenmodell sowohl die Syntax der Methoden, als auch deren Semantik hinterlegt.

Für die Biathlondomäne könnte die Menge der Schnittstellen beispielsweise wie folgt definiert sein:

DomainArchitecture = {BiathlonDomain}
DomainInterface = {ITrainer, IAthlete, IPulse}
defines(BiathlonDomain) = {ITrainer, IAthlete, IPulse}

Wie später noch erläutert wird, können Komponenten Dienste anbieten und Dienste anderer Komponenten nutzen. Jeder angebotene Dienst implementiert hierbei genau eine Domänenschnittstelle. Über die implementierte Domänenschnittstelle können später potentielle Dienstnutzer geeignete Dienste finden. Möchte beispielsweise eine Komponente den aktuellen Puls eines Sportlers als Dienst anderen Komponenten bereitstellen, so würde er sich in der Domänenarchitektur auf die Domänenschnittstelle *IPulse* beziehen und diese implementieren. Eine Sportlerkomponente, welche den aktuellen Puls anzeigen möchte, bezieht sich ebenfalls auf diese Schnittstelle und kennt somit die Methode, mit der der Puls abgefragt werden kann. Für die Sportlerkomponente ist es hierbei zunächst unerheblich, von welchem Dienst diese Schnittstelle implementiert wird.

4.4 Spezifikation von Komponenten

Jede Komponente (Komponenteninstanz) wird im Komponentenmodell durch genau eine Instanz der Klasse *DynamicAdaptiveComponent* repräsentiert. Die Attribute und Methoden dieser Klasse werden im nächsten Abschnitt detailliert erläutert.

4.4.1 Attribute und Methoden der Klasse *DynamicAdaptiveComponent*

Die Klasse besitzt ein Attribut sowie fünf Methoden und ist in Abbildung 4-7 nochmals dargestellt.

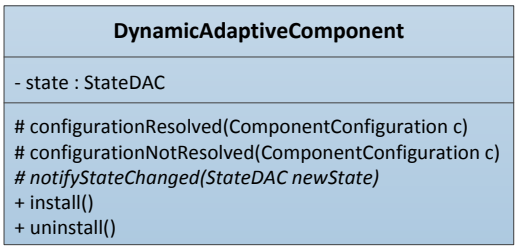


Abbildung 4-7: Die Klasse *DynamicAdaptiveComponent*

Jede Instanz von *DynamicAdaptiveComponent* realisiert einen Zustandsautomaten. Dieser besteht aus zwei Zuständen, nämlich NOT_RESOLVED und RESOLVED. Der aktuelle Zustand des Automaten wird im Attribut *state* hinterlegt (siehe Formel 4-2):

$$\boldsymbol{StateDAC} = \{NOT\_RESOLVED, RESOLVED\}$$
$$\boldsymbol{state} \in StateDAC$$

Formel 4-2

Ein Zustandsübergang innerhalb des Automaten findet immer dann statt, wenn eine der Methoden *configurationResolved*, *configurationNotResolved*, *install* oder *uninstall* aufgerufen wird. Hierbei handelt es sich um sogenannte Trigger, die entweder durch andere Klassen des Komponentenmodells oder durch eine Komponentenimplementierung aufgerufen werden. Der zugehörige Zustandsautomat wird in Abschnitt 4.4.3 im Detail erläutert.

Die Methoden *install* und *uninstall* dienen dazu, das Framework zu veranlassen, die Komponente sowie deren Dienste anderen Komponenten bekannt zu machen und mit der Auflösung von Abhängigkeiten zu beginnen, bzw. die Ausführung der Komponente zu beenden. Diese Methoden werden vom Framework implementiert und von der jeweiligen Komponente im Bedarfsfall aufgerufen. So können beispielsweise innerhalb einer Komponente zunächst Initialisierungsmaßnahmen durchgeführt werden, bevor die Komponente und deren angebotenen Dienste durch Aufruf von *install* verfügbar gemacht werden. Der Trigger *uninstall* könnte durch eine Komponente beispielsweise dann aufgerufen werden, wenn das Gerät, auf dem die Komponente installiert ist, ausgeschaltet wird.

Die Methode *notifyStateChanged* wird hingegen nicht vom Framework implementiert. Es handelt sich hierbei um eine abstrakte Methode, welche durch das Framework aufgerufen wird und vom Komponentenentwickler implementiert werden muss. Diese Methode wird vom Framework immer dann aufgerufen, wenn ein Zustandswechsel innerhalb von *DynamicAdaptiveComponent* stattfindet. Die Komponente kann durch Überschreiben der Methode *notifyStateChanged* dann ggf. auf einen Zustandsübergang reagieren.

Bei den Methoden *configurationResolved* und *configurationNotResolved* handelt es sich um Trigger, welche der frameworkinternen Kommunikation zwischen der *DynamicAdaptiveComponent* und anderen Elementen des Komponentenframeworks dienen. Über die Methode *configurationResolved* teilt eine *ComponentConfiguration* der

DynamicAdaptiveComponent mit, dass sämtliche Abhängigkeiten für diese Komponentenkonfiguration aufgelöst werden konnten. Die Methode *configurationNotResolved* wird von einer *ComponentConfiguration* hingegen immer dann aufgerufen, wenn die Abhängigkeiten nicht mehr aufgelöst sind. Durch diese Trigger wird die *DynamicAdaptiveComponent* über die Menge der aktivierbaren Konfigurationen auf dem Laufenden gehalten. Diese bewirken ggf. auch Zustandsübergänge im Automaten der Komponente. Die Methoden werden im Rahmen der Vorstellung des Zustandsautomaten von *DynamicAdaptiveComponent* in Abschnitt 4.4.3 näher erläutert.

4.4.2 Beziehungen zu anderen Klassen des Komponentenmodells

Wie bereits in Abschnitt 4.2 erläutert wurde, bietet eine *DynamicAdaptiveComponent* für sich betrachtet weder Dienste an, noch benötigt sie Dienste anderer Komponenten. Stattdessen werden Dienste über *ComponentConfigurations* angeboten. Diese definieren gleichzeitig Abhängigkeiten zu Diensten anderer Komponenten. Eine Komponente kann beliebig viele solcher *ComponentConfigurations* definieren (siehe Formel 4-3):

$$\boldsymbol{contains}: DynamicAdaptiveComponent \rightarrow \mathcal{P}(ComponentConfiguration)$$

Formel 4-3

Konnten für eine *ComponentConfiguration* alle definierten Abhängigkeiten aufgelöst werden, so werden die entsprechenden angebotenen Dienste jedoch nicht sofort gestartet, sondern es wird zunächst die zugehörige *DynamicAdaptiveComponent* durch Aufruf von *configurationResolved* hierüber informiert. Eine *ComponentConfiguration* wird dann als aktivierbar bezeichnet. Die Menge aller zu einem Zeitpunkt aktivierbaren Komponentenkonfigurationen ist im Modell über die Assoziation *activatable* repräsentiert und in Formel 4-4 formalisiert angegeben:

$$\boldsymbol{activatable}: DynamicAdaptiveComponent \rightarrow \mathcal{P}(ComponentConfiguration)$$
$$\forall dac \in DynamicAdaptiveComponent: activatable(dac) \subseteq contains(dac)$$

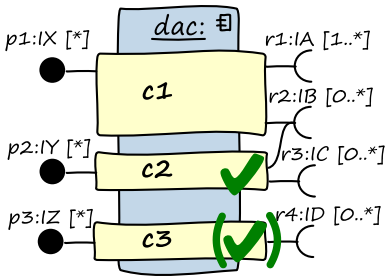
Formel 4-4

Das hier vorgestellte Komponentenmodell sieht vor, dass zu jedem Zeitpunkt maximal eine *ComponentConfiguration* aktiv ist. Diese ist im Modell durch die Assoziation *current* repräsentiert und in Formel 4-5 angegeben:

$$\boldsymbol{current}: DynamicAdaptiveComponent \rightarrow ComponentConfiguration$$
$$\forall dac \in DynamicAdaptiveComponent: activatable(dac): current(dac) \in activatable(dac)$$

Formel 4-5

Die Aufgabe des Frameworks ist es, für eine *DynamicAdaptiveComponent* die Menge der aktivierbaren *ComponentConfigurations* zu verwalten und die beste hieraus auszuwählen. Ist mehr als eine Komponentenkonfiguration aktivierbar, so sind verschiedene Varianten denkbar, hieraus diejenige zu bestimmen, welche ausgeführt werden soll. Im vorliegenden Komponentenmodell definiert die Komponente eine totale Ordnung über alle *ComponentConfigurations* der Menge *contains*. Das Framework wählt dann stets die bestmögliche Konfiguration dieser geordneten Menge. Anhand der in Abbildung 4-8 dargestellten Komponente kann die Funktionsweise verdeutlicht werden.



$$contains(dac) = \{c_1, c_2, c_3\}$$
$$activatable(dac) = \{c_2, c_3\}$$
$$current(dac) = c_2$$

Abbildung 4-8: Beispiel einer Komponente mit drei *ComponentConfigurations*, davon zwei aktivierbar und eine aktiv

Die Komponente besitzt drei *ComponentConfigurations*, wobei die Komponentenkonfiguration c_1 die beste ist. Diese kann allerdings im Gegensatz zu den anderen beiden nicht aktiviert werden, da sie ein *RequiredServiceReferenceSet* besitzt, für welches die Abhängigkeiten nicht aufgelöst werden konnten. In diesem Fall wird die Komponentenkonfiguration c_2 aktiviert, da diese in der Ordnung vor c_3 steht. Für beide konnten sämtliche Abhängigkeiten zu Diensten anderer Komponenten aufgelöst werden.

4.4.3 Der Zustandsautomat von *DynamicAdaptiveComponent*

Wie bereits zuvor erwähnt, ist die Hauptaufgabe einer *DynamicAdaptiveComponent*, die bestmögliche Komponentenkonfiguration zu finden und zu aktivieren. Sie realisiert zu diesem Zweck einen Zustandsautomaten, welcher die Zustände NOT_RESOLVED und RESOLVED unterscheidet. Eine Komponente befindet sich im Zustand NOT_RESOLVED, solange sie keine aktive *ComponentConfiguration* besitzt. Sobald eine aktivierbare Komponentenkonfiguration verfügbar ist, wechselt die Komponente in den Zustand RESOLVED.

Der Zustandsautomat sowie die zugehörigen Übergangsbedingungen und Trigger sind in Abbildung 4-9 in Form eines UML Zustandsdiagramms [Obj09] dargestellt.

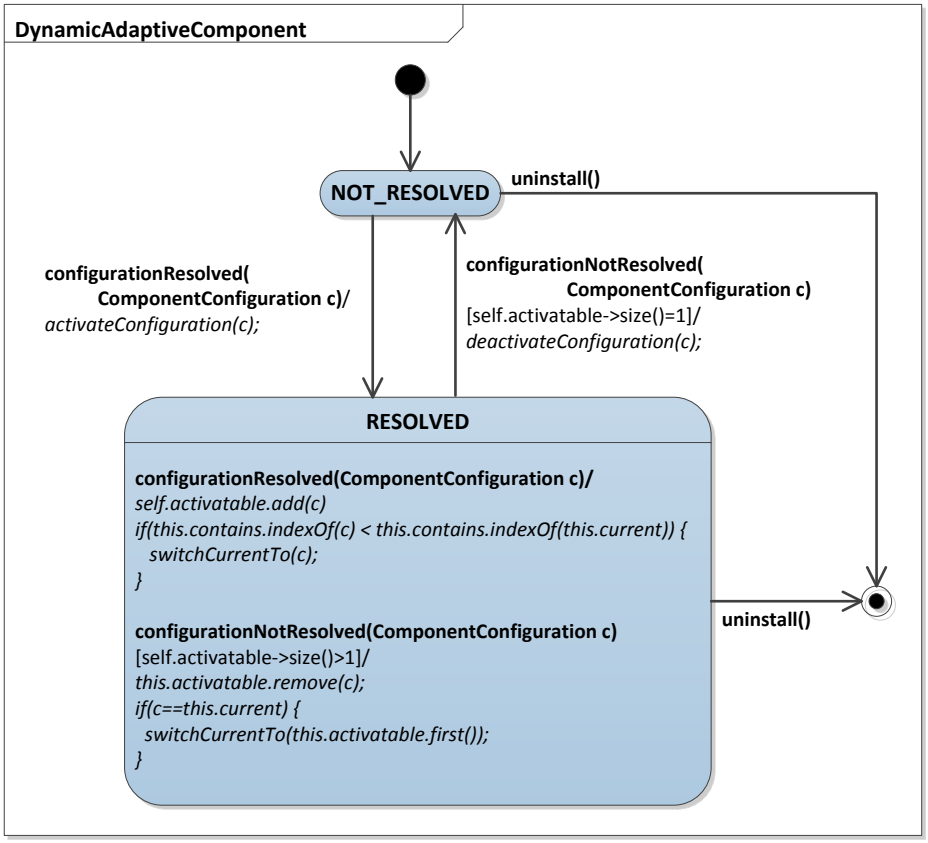


Abbildung 4-9: Der Zustandsautomat von *DynamicAdaptiveComponent*

In UML-Zustandsdiagrammen wird der Startzustand als ausgefüllter Kreis dargestellt, und Endzustände als ausgefüllter Kreis innerhalb eines Kreises. Alle anderen Zustände werden als abgerundete Rechtecke abgebildet. Jeder Zustand besitzt einen eindeutigen Bezeichner (z.B. NOT_RESOLVED oder RESOLVED). Zustandsübergänge werden mit Pfeilen notiert. Diese Pfeile sind jeweils mit einem Trigger beschriftet, welcher den Zustandsübergang auslöst. Im Folgenden werden die Trigger aus Gründen der Übersichtlichkeit in Fettschrift notiert. Ein Zustandsübergang kann zusätzlich an bestimmte Bedingungen geknüpft sein, sogenannte Guards. Diese werden hinter dem Trigger in eckigen Klammern notiert. Zur Spezifikation dieser Guards wird in dieser Arbeit OCL verwendet [Obj06]. Ein Zustandsübergang findet in diesem Fall nur dann statt, falls der Trigger aufgerufen wird und die angegebenen Bedingungen erfüllt sind. Und schließlich können zu einem Trigger eine oder mehrere Aktionen definiert werden, die im Falle des Auftretens des entsprechenden Triggers und unter der Voraussetzung, dass der Guard erfüllt ist, ausgeführt werden. Diese Aktionen werden im Folgenden als Javacode beschrieben und kursiv dargestellt. Hierbei werden die Aktionen häufig direkt im Zustandsautomaten angegeben. In einigen Fällen sind aus Gründen der Übersichtlichkeit Aktionen zu Methoden zusammengefasst, welche dann im Text angegeben sind. Es werden nun die einzelnen Zustände sowie die abgehenden Zustandsübergänge im Detail erläutert.

Der Zustand NOT_RESOLVED

Im Zustand NOT_RESOLVED besitzt die Komponente keine *ComponentConfiguration*, für die alle Abhängigkeiten aufgelöst werden konnten. Erst sobald die *DynamicAdaptiveComponent* von einer der Konfigurationen durch Aufruf des Triggers *configurationResolved* mitgeteilt bekommt, dass eine Konfiguration aktivierbar geworden ist, geht die Komponente in den Zustand

RESOLVED über. Die entsprechende *ComponentConfiguration* wird in diesem Fall der Menge *activatable* hinzugefügt. Außerdem wird diese Konfiguration automatisch zur aktiven Konfiguration, da es sich zunächst um die einzige aktivierbare Konfiguration handelt. Alle *ProvidedServices*, welche von dieser Konfiguration angeboten werden, werden außerdem durch Aufruf von *serviceRunnable* darüber informiert, dass sie ausgeführt werden können. Wie später gezeigt wird, führt das nicht automatisch auch zu einer Ausführung der entsprechenden Dienste. Hierzu muss zusätzlich entweder das Flag *requestRun* gesetzt sein, oder es müssen Verwender des Dienstes in Form von anderen Komponenten existieren.

Die prinzipielle Ausführbarkeit wird den *ProvidedServices* durch Aufruf des Triggers *serviceRunnable* mitgeteilt. Die Entscheidung, ob der Dienst dann auch ausgeführt wird, wird innerhalb des *ProvidedServices* entschieden. Die bei einem Zustandsübergang von NOT_RESOLVED nach RESOLVED durchzuführenden Aktionen wurden in der Methode *activateConfiguration* zusammengefasst, welche in Listing 4-1 dargestellt ist.

```
1  activateConfiguration(ComponentConfiguration c)  {
2   this.activatable.add(c);
3   this.current = c;
4   for(ProvidedService ps : c.provides) {
5    ps.serviceRunnable();
6   }
7  }
```

Listing 4-1: Die Methode *activateConfiguration*

Im Wesentlichen wird also die entsprechende Konfiguration der Menge *activatable* hinzugefügt, als *current* markiert und die angebotenen Dienste der Konfiguration über deren Ausführbarkeit informiert.

Wird die Komponente durch Aufruf von *uninstall* deinstalliert, so werden alle Bestandteile der Komponente (*ComponentConfigurations, ProvidedServices* und *RequiredServiceReferenceSets*) hierüber durch Aufruf von *uninstall* informiert. Diese sorgen dann für eine saubere Beendigung ihrer Dienste.

Der Zustand RESOLVED

Im Zustand RESOLVED besitzt die Komponente mindestens eine *ComponentConfiguration* mit aufgelösten Abhängigkeiten. In dieser Situation können nun entweder neue aktivierbare Konfiguration hinzukommen, oder aktivierbare Konfigurationen ausfallen, da deren Abhängigkeiten nicht mehr aufgelöst sind.

Kommt eine weitere aktivierbare *ComponentConfiguration* hinzu, so wird das der *DynamicAdaptiveComponent* über den Trigger *configurationResolved* mitgeteilt. In diesem Fall wird dann zunächst geprüft, ob die neue Konfiguration besser ist als die aktuell aktive Konfiguration. Ist dies der Fall, so können nur noch diejenigen Dienste ausgeführt werden, welche auch von der neuen Konfiguration angeboten werden. Alle anderen Dienste können nicht mehr ausgeführt werden, was diesen Diensten durch Aufruf des Triggers *serviceNotRunnable* mitgeteilt wird. Die notwendigen Aktionen für den Austausch der aktiven Konfiguration wurden in der Methode *switchCurrentTo* zusammengefasst. Die Methode ist in Listing 4-2 abgebildet.

```
1  switchCurrentTo(ComponentConfiguration c)  {
2   for(ProvidedService ps : c.provides) {
3    if(!this.current.provides.contains(ps)) {
4     ps.serviceRunnable();
5    }
6   }
7   for(ProvidedService ps : this.current.provides) {
8    if(!c.provides.contains(ps)) {
9     ps.serviceNotRunnable();
10   }
11  }
12 }
13 this.current = c;
```

Listing 4-2: Die Methode *switchCurrentTo*

Ist eine *ComponentConfiguration* nicht mehr aktivierbar, so wird das der *DynamicAdaptiveComponent* über den Trigger *configurationNotResolved* mitgeteilt. Auf diesen Trigger können zwei unterschiedliche Reaktionen erfolgen. Handelt es sich bei der Komponentenkonfiguration um die letzte aktivierbare Konfiguration, so kann kein Dienst der Komponente mehr ausgeführt werden und sie wechselt zurück in den Zustand NOT_RESOLVED. Die hierbei auszuführenden Aktionen sind in der Methode *deactivateConfiguration* zusammengefasst, welche in Listing 4-3 dargestellt ist.

```
1  deactivateConfiguration(ComponentConfiguration c)  {
2   for(ProvidedService ps : c.provides) {
3    ps.serviceNotRunnable();
4   }
5   this.activatable.remove(c);
6   this.current = null;
7  }
```

Listing 4-3: Die Methode *deactivateConfiguration*

Existieren allerdings noch weitere aktivierbare Konfigurationen innerhalb der Menge *activatable*, und handelt es sich bei der wegfallenden Konfiguration um die aktive Konfiguration, so findet ein Wechsel der aktiven Konfiguration statt.

Wird aus dem Zustand RESOLVED heraus *uninstall* aufgerufen, so werden wie bei NOT_RESOLVED sämtliche Elemente der Komponente durch Aufruf von *uninstall* hierüber informiert. Diese beenden daraufhin ihre Dienste und gehen ebenfalls in den Endzustand über.

4.5 Spezifikation von Komponentenkonfigurationen

Wie bereits zuvor angedeutet wurde, handelt es sich bei *ComponentConfigurations* um Abbildungen von angebotenen Diensten auf benötigte Dienste einer Komponente. Wie später gezeigt wird, können angebotene Dienste einer Komponente nur dann ausgeführt werden, wenn sie mit mindestens einer *ComponentConfiguration* verbunden sind, für die alle Abhängigkeiten zu Diensten anderer Komponenten aufgelöst werden konnten. Eine Komponente besteht aus einer oder mehreren *ComponentConfigurations*. Eine *ComponentConfiguration* kann sowohl mit angebotenen Diensten (*ProvidedServices*) als auch mit benötigten Diensten (*RequiredServiceReferenceSets)* verbunden sein. Letztere definieren Abhängigkeiten zu angebotenen Diensten anderer Komponenten.

4.5.1 Attribute und Methoden der Klasse *ComponentConfiguration*

Die Klasse *ComponentConfiguration* definiert ein Attribut sowie sieben Methoden und ist in Abbildung 4-10 nochmals dargestellt.

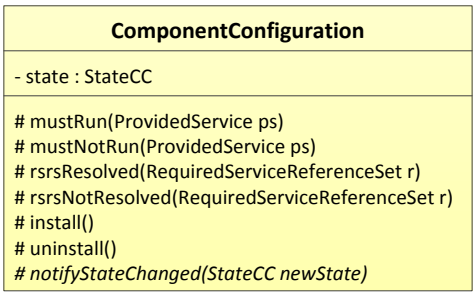


Abbildung 4-10: Die Klasse *ComponentConfiguration*

Genauso wie die Klasse *DynamicAdaptiveComponent* realisiert auch die Klasse *ComponentConfiguration* einen eigenen Zustandsautomaten, dessen aktueller Zustand im Attribut *state* vom Typ *StateCC* abgelegt ist. Der Zustandsautomat von *ComponentConfiguration* definiert drei Zustände, nämlich NOT_RESOLVED, RESOLVING und RESOLVED (siehe Formel 4-6):

$$\boldsymbol{StateCC} = \{NOT\_RESOLVED, RESOLVING, RESOLVED\}$$
$$\boldsymbol{state} \in StateCC$$

Formel 4-6

Befindet sich eine *ComponentConfiguration* im Zustand NOT_RESOLVED, so sind dessen Abhängigkeiten zu anderen Komponenten weder aufgelöst, noch wird versucht, sie aufzulösen. Im Zustand RESOLVING werden Anbieter von Diensten gesucht, welche durch die *ComponentConfiguration* benötigt werden. Sobald alle Abhängigkeiten einer *ComponentConfiguration* aufgelöst werden konnten, befindet sie sich im Zustand RESOLVED. Der zugehörige Zustandsautomat wird in Abschnitt 4.5.3 detailliert vorgestellt.

Der Zustandsautomat reagiert, wie die *DynamicAdaptiveComponent* auch, auf den Aufruf von Trigger. Der Zustandsautomat von *ComponentConfiguration* definiert hierzu sechs Trigger: *mustRun*, *mustNotRun*, *rsrsResolved*, *rsrsNotResolved*, *install* und *uninstall*. Durch die Trigger *mustRun* bzw. *mustNotRun* signalisieren *ProvidedServices*, ob sie ausgeführt werden müssen. Die *ComponentConfiguration* beginnt bzw. beendet daraufhin das Auflösen von Abhängigkeiten. Über die Trigger *rsrsResolved* bzw. *rsrsNotResolved* wird die Komponentenkonfiguration von verbundenen *RequiredServiceReferenceSets* darüber informiert, ob die Abhängigkeiten aufgelöst sind oder nicht. Über die Trigger *install* wird die Ausführung des Zustandsautomaten gestartet, und durch *uninstall* beendet. Der vollständige Automat und damit auch die Funktionsweise der Trigger wird ausführlich in Abschnitt 4.5.3 beschrieben.

Jede *ComponentConfiguration* definiert, genau wie *DynamicAdaptiveComponent* auch, eine abstrakte Methode namens *notifyStateChanged*, über die das Framework die Komponente über einen Zustandswechsel informiert. Durch Überschreiben dieser Methode kann die Komponente auf Zustandswechsel innerhalb des Automaten reagieren.

4.5.2 Beziehungen zu anderen Klassen des Komponentenmodells

Wie bereits in Abschnitt 4.4 beschrieben wurde, muss eine *DynamicAdaptiveComponent* stets über die Menge der aktivierbaren *ComponentConfigurations* auf dem Laufenden gehalten werden. Dies geschieht durch Aufruf der Trigger *configurationResolved* bzw. *configurationNotResolved*. Aus diesem Grunde benötigt jede *ComponentConfiguration* eine Referenz auf die ihr zugeordnete Komponente. Hierbei ist jede Komponentenkonfiguration genau einer Komponente zugeordnet (siehe Formel 4-7):

$$\boldsymbol{containedBy}: ComponentConfiguration \rightarrow DynamicAdaptiveComponent$$

Formel 4-7

Wie bereits erwähnt, definiert jede *ComponentConfiguration* eine Abbildung von benötigten Diensten auf angebotene Dienste einer Komponente. Die mit einer Konfiguration verknüpften *ProvidedServices* und *RequiredServiceReferenceSets* sind über die die Assoziation *provides* bzw. *declares* verbunden. Jede Konfiguration kann hierbei mit beliebig vielen *ProvidedServices* und *RequiredServiceReferenceSets* verbunden sein. Die Formalisierung dieser Assoziationen ist in Formel 4-8 angegeben:

$$\boldsymbol{provides}: ComponentConfiguration \rightarrow \mathcal{P}(ProvidedService)$$
$$\boldsymbol{declares}: ComponentConfiguration \rightarrow \mathcal{P}(RequiredServiceReferenceSet)$$

Formel 4-8

Nachdem eine Komponente und deren Bestandteile (*ComponentConfigurations*, *ProvidedServices* und *RequiredServiceReferenceSets*) beim Framework installiert sind, wird nicht unmittelbar damit begonnen, Abhängigkeiten zu Diensten anderer Komponenten aufzulösen. Vielmehr wird darauf gewartet, dass der Bedarf zur Ausführung einer der angebotenen Dienste besteht. Dieser besteht beispielsweise immer dann, wenn Nutzungsanfragen anderer Komponenten für einen Dienst existieren. Nun kann eine *ComponentConfiguration* beliebig viele Dienste anbieten, von denen zu einem Zeitpunkt ggf. einige ausgeführt werden müssen, und andere nicht. Eine Komponentenkonfiguration muss hierbei also stets wissen, ob sie mit auszuführenden *ProvidedServices* verknüpft ist. Sie wird zu diesem Zweck über die bereits vorgestellten Methoden *mustRun* bzw. *mustNotRun* durch *ProvidedServices* auf dem Laufenden gehalten. Die Menge der auszuführenden Dienste einer *ComponentConfiguration* wird im Komponentenmodell aus Abbildung 4-5 über die Assoziation *runRequestedBy* repräsentiert (siehe Formel 4-9):

$$\boldsymbol{runRequestedBy}: ComponentConfiguration \rightarrow \mathcal{P}(ProvidedService)$$
$$\forall c \in ComponentConfiguration:$$
$$runRequestedBy(c) \subseteq provides(c)$$

Formel 4-9

Die Abhängigkeiten, die die jeweilige *ComponentConfiguration* definieren, werden nur dann aufgelöst, wenn die Menge *runRequestedBy* nicht leer ist, also ein Bedarf für die Ausführung mindestens eines Dienstes besteht.

Auf der anderen Seite muss jede *ComponentConfiguration* darüber auf dem Laufenden gehalten werden, inwieweit Abhängigkeiten zu Diensten anderer Komponenten aufgelöst werden konnten. Solche Abhängigkeiten werden durch *RequiredServiceReferenceSets* definiert. Diese melden durch Aufruf der bereits vorgestellten Trigger *rsrsResolved* bzw. *rsrsNotResolved*, ob die

durch ein *RequiredServiceReferenceSet* definierten Abhängigkeiten aufgelöst sind oder nicht. Diejenigen mit einer *ComponentConfiguration* verknüpften *RequiredServiceReferenceSets*, deren Abhängigkeiten aufgelöst sind, werden im Modell durch die Assoziation *resolved* repräsentiert (siehe Formel 4-10):

$$\boldsymbol{resolved}: ComponentConfiguration \rightarrow \mathcal{P}(RequiredServiceReferenceSet)$$
$$\forall c \in ComponentConfiguration:$$
$$resolved(c) \subseteq declares(c)$$

Formel 4-10

Im nun folgenden Abschnitt wird der Zustandsautomat von *ComponentConfiguration* vorgestellt. Hierbei wird u.a. auch die Bedeutung der einzelnen bereits vorgestellten Trigger und Assoziationen nochmals verdeutlicht.

4.5.3 Der Zustandsautomat von *ComponentConfiguration*

Eine der wesentlichen Aufgaben einer *ComponentConfiguration* während des Konfigurationsprozesses ist es, zwischen angebotenen und benötigten Diensten der Komponentenkonfiguration zu vermitteln und die *DynamicAdaptiveComponent* bzgl. dessen Aktivierbarkeit informieren. Teilt beispielsweise ein angebotener Dienst einer *ComponentConfiguration* über den Trigger *mustRun* mit, dass er ausgeführt werden muss, so informiert sie alle verbundenen *RequiredServiceReferenceSets* hierüber. Diese beginnen dann mit der Auflösung der Abhängigkeiten. Auf der anderen Seite verwaltet eine *ComponentConfiguration* die Menge der bereits aufgelösten *RequiredServiceReferenceSets*. Sobald alle Abhängigkeiten aller *RequiredServiceReferenceSets* der Konfiguration aufgelöst werden konnten, wird die Komponente hierüber informiert. Diese wiederum bestimmt dann, ob die Konfiguration auch aktiviert wird. Letzteres wurde bereits im Abschnitt 4.4 beschrieben. Zur Realisierung dieser unterschiedlichen Aufgaben realisiert jede *ComponentConfiguration* einen Zustandsautomaten mit den Zuständen NOT_RESOLVED, RESOLVING und RESOLVED. Der entsprechende Zustandsautomat ist in Abbildung 4-11 in Form eines UML-Zustandsdiagramms dargestellt.

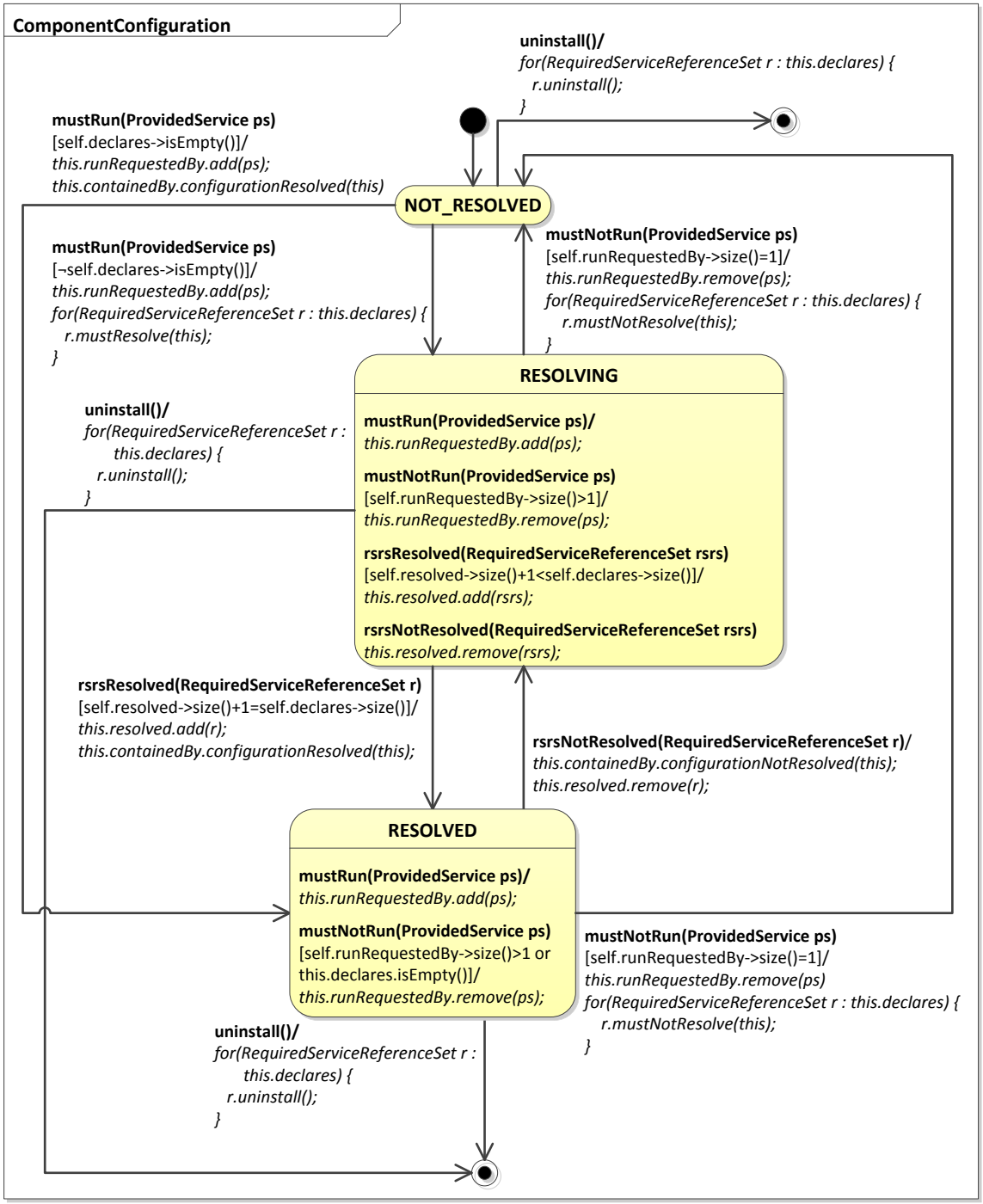


Abbildung 4-11: Zustandsautomat von *ComponentConfiguration*

Im Folgenden werden die einzelnen Zustände sowie die jeweils möglichen Zustandsübergänge erläutert.

Der Zustand NOT_RESOLVED

Nachdem eine *DynamicAdaptiveComponent* installiert wurde, werden alle *ComponentConfigurations* in den Zustand NOT_RESOLVED versetzt. In diesem Zustand sind ggf. definierte Abhängigkeiten zu Diensten anderer Komponenten weder aufgelöst, noch versucht das Framework, sie aufzulösen. Mit dem Auflösen von Abhängigkeiten wird erst dann begonnen, wenn die *ComponentConfiguration* einen Dienst anbietet, welcher ausgeführt werden muss.

ProvidedServices teilen dies den *ComponentConfigurations* durch Aufruf des Triggers *mustRun* mit. Diese Dienste werden in diesem Fall der Menge *runRequestedBy* hinzugefügt.

Sobald der Trigger *mustRun* bei einer Komponentenkonfiguration aufgerufen wurde, beginnt das Framework mit der Auflösung ggf. definierten Abhängigkeiten zu Diensten anderer Komponenten. Derartige Abhängigkeiten werden durch *RequiredServiceReferenceSets* definiert. Definiert eine *ComponentConfiguration* kein *RequiredServiceReferenceSet*, so gelten die Abhängigkeiten für diese Komponentenkonfiguration als aufgelöst. Sie wechselt daraufhin unmittelbar in den Zustand RESOLVED.

Für den Fall, dass eine *ComponentConfiguration* ein oder mehrere *RequiredServiceReferenceSets* definiert, kann sie bei Aufruf des Triggers *mustRun* nicht unmittelbar in den Zustand RESOLVED wechseln, sondern geht stattdessen zunächst in den Zustand RESOLVING über. Bei diesem Zustandswechsel werden gleichzeitig alle mit der Komponentenkonfiguration verknüpften *RequiredServiceReferenceSets* durch Aufruf von *mustResolve* dazu aufgefordert, die durch sie definierten Abhängigkeiten aufzulösen. Eine *ComponentConfiguration* verbleibt solange in diesem Zustand, bis alle verbundenen *RequiredServiceReferenceSets* über den Trigger *rsrsResolved* gemeldet haben, dass die Abhängigkeiten aufgelöst werden konnten. Jedes *RequiredServiceReferenceSet*, welches diese Rückmeldung gibt, wird der Menge *resolved* hinzugefügt. Auf diese Weise kann später festgestellt werden, wann alle Abhängigkeiten für eine *ComponentConfiguration* aufgelöst werden konnten.

Der Zustand RESOLVING

In den Zustand RESOLVING wechselt eine *ComponentConfiguration* nur, wenn sie *RequiredServiceReferenceSets* definiert. Eine Komponentenkonfiguration wartet in diesem Zustand darauf, dass jedes *RequiredServiceReferenceSet* über den Trigger *rsrsResolved* gemeldet hat, dass die Abhängigkeiten aufgelöst werden konnten. Die Menge der bereits aufgelösten *RequiredServiceReferenceSets* ist im Modell durch die Assoziation *resolved* repräsentiert. Immer dann, wenn ein *RequiredServiceReferenceSet* meldet, dass die Abhängigkeiten aufgelöst werden konnten, wird die Menge *resolved* entsprechend ergänzt. Bei Aufruf des Triggers *rsrsNotResolved* wird aus der Menge das entsprechende Element entfernt. Sobald alle *RequiredServiceReferenceSets* gemeldet haben, dass ihre Abhängigkeiten aufgelöst werden konnten, geht die *ComponentConfiguration* in den Zustand RESOLVED über und die *DynamicAdaptiveComponent* wird hierüber informiert. Diese nimmt die Komponentenkonfiguration daraufhin in die Menge *activatable* auf und aktiviert sie gegebenenfalls (siehe Abschnitt 4.4**Fehler! Verweisquelle konnte nicht gefunden werden.**).

Gibt es keine Nutzungsanfragen mehr für mindestens einen angebotenen Dienst der *ComponentConfiguration*, so wechselt sie zurück in den Zustand NOT_RESOLVED. Die verbundenen *RequiredServiceReferenceSets* werden in diesem Fall durch Aufruf von *mustNotResolve* außerdem darüber informiert, dass die *ComponentConfiguration* kein Interesse mehr an der Auflösung der Abhängigkeiten hat.

Der Zustand RESOLVED

Befindet sich eine *ComponentConfiguration* im Zustand RESOLVED, so sind alle Abhängigkeiten aufgelöst und die angebotenen Dienste der Konfiguration können ausgeführt werden. Außerdem ist diese Komponentenkonfiguration Teil der Menge *activatable* der zugehörigen *DynamicAdaptiveComponent*. Wie zuvor beschrieben, kümmert sich anschließend die

DynamicAdaptiveComponent darum, den angebotenen Diensten der bestmöglichen aktivierbaren *ComponentConfiguration* mitzuteilen, ob sie ausgeführt werden sollen oder nicht.

Im Zustand RESOLVED nimmt eine *ComponentConfiguration* weiterhin Nutzungsanfragen bzw. deren Rücknahme (*mustRun* bzw. *mustNotRun*) entgegen. Immer wenn eine Nutzungsanfrage entfällt, so wird geprüft, ob noch weitere Anfragen vorliegen, also die Menge *runRequestedBy* leer ist oder nicht. Sobald keine Nutzungsanfragen mehr vorliegen, wechselt die Konfiguration zurück in den Zustand NOT_RESOLVED und die *DynamicAdaptiveComponent* sowie alle verbundenen *RequiredServiceReferenceSets* werden hierüber informiert.

Sobald ein verbundenes *RequiredServiceReferenceSet* meldet, dass die Abhängigkeiten nicht mehr aufgelöst werden konnten, wechseln alle betroffenen *ComponentConfigurations* in den Zustand RESOLVING. Hierüber wird die *DynamicAdaptiveComponent* informiert. Da ggf. weiterhin Nutzungsanfragen vorliegen, wird weiterhin versucht, die Komponentenkonfiguration in einen aktivierbaren Zustand zu versetzen.

In den folgenden zwei Abschnitten werden nun noch die Elemente *ProvidedService* und *RequiredServiceReferenceSet* vorgestellt, bevor dann genauer auf die Assoziationen zwischen diesen beiden Elementen eingegangen wird.

4.6 Spezifikation angebotener Dienste

Im vorliegenden Komponentenmodell wird Funktionalität von Komponenten über angebotene Dienste anderen Komponenten oder Endanwendern zur Verfügung gestellt. Diese angebotenen Dienste sind im Komponentenmodell aus Abbildung 4-5 repräsentiert durch die Klasse *ProvidedService*. Im folgenden Abschnitt werden zunächst die Attribute und Methoden der Klasse näher erläutert, gefolgt von der Einordnung in das Framework sowie der Vorstellung des Zustandsautomaten, welcher durch die Klasse realisiert wird.

4.6.1 Attribute und Methoden der Klasse *ProvidedService*

Die Klasse *ProvidedService* aus Abbildung 4-5 ist in Abbildung 4-12 nochmals dargestellt.

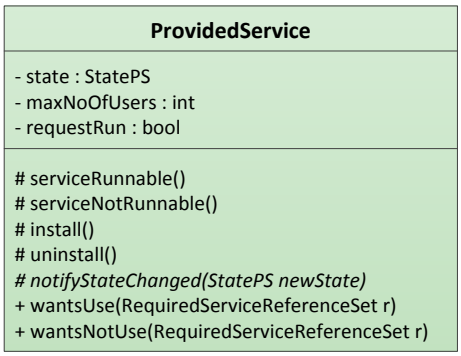


Abbildung 4-12: Die Klasse *ProvidedService*

Sie definiert drei Attribute sowie sieben Methoden, welche im Folgenden näher vorgestellt werden.

Wie *DynamicAdaptiveComponent* und *ComponentConfiguration* realisiert auch *ProvidedService* einen Zustandsautomaten. Der aktuelle Zustand wird im Attribut *state* vom Typ *StatePS*

gehalten. Der Zustandsautomat besteht aus drei Zuständen, nämlich NOT_RUNNING, RUNNABLE und RUNNING (siehe Formel 4-11):

$$\boldsymbol{StatePS} = \{NOT\_RUNNING, RUNNABLE, RUNNING\}$$
$$\boldsymbol{state} \in StatePS$$

Formel 4-11

Im Zustand NOT_RUNNING wird der Dienst nicht ausgeführt und kann somit nicht verwendet werden. Diesen Zustand nimmt jeder Dienst unmittelbar nach Installation der Komponente an. Befindet sich ein Dienst im Zustand RUNNABLE, so kann der Dienst prinzipiell ausgeführt werden. Er läuft allerdings noch nicht und ist somit ebenfalls nicht verwendbar. Dieser Fall tritt z.B. dann auf, wenn alle Abhängigkeiten für diesen Dienst durch eine *ComponentConfiguration* aufgelöst werden konnten, aber für diesen Dienst keine Notwendigkeit für eine Ausführung besteht. Erst, wenn sich ein Dienst im Zustand RUNNING befindet, kann er durch andere Komponenten oder Endanwender verwendet werden. Hierbei ist der Dienst mit einer aktiven Konfiguration verbunden und es besteht zusätzlich die Notwendigkeit für eine Ausführung. Die Notwendigkeit zur Ausführung eines Dienstes kann, wie später noch näher erläutert wird, zwei Ursachen haben. Zum einen liegt die Notwendigkeit dann vor, wenn Nutzungsanforderungen in Form von anderen Komponenten existieren. Zum anderen liegt die Notwendigkeit vor, wenn das Flag *requestRun* gesetzt ist. Dieses Attribut der Klasse *ProvidedService* wird im Folgenden erläutert.

Das Flag *requestRun* vom Typ *bool* nimmt Einfluss darauf, unter welchen Umständen der Dienst ausgeführt werden soll (siehe Formel 4-12):

$$\boldsymbol{requestRun}: ProvidedService \rightarrow bool$$

Formel 4-12

Im Framework werden hierbei zwei Arten von Diensten unterschieden. So gibt es zum einen Dienste, deren Ausführung nur dann sinnvoll ist, wenn diese durch andere Komponenten verwendet werden möchten. Ein Beispiel hierfür ist beispielsweise eine Pulssensorkomponente. Über dessen angebotenen Dienst kann lediglich der aktuelle Puls über die Methode *getPulse* abgefragt werden. Dieser Dienst alleine hat für Endanwender zunächst keinen Nutzen. Erst, wenn eine weitere Komponente zur Verfügung steht, welche den Puls weiterverarbeitet oder dem Nutzer beispielsweise den Puls anzeigt, ist eine Ausführung des Dienstes sinnvoll. Die Ausführung des Pulsdienstes ohne die Verfügbarkeit potentieller Nutzer kann zudem zu unnötigem Stromverbrauch führen. Derartige Dienste, welche für sich genommen keinerlei Nutzen erbringen, können dem Framework durch das Setzen des Flags *requestRun* auf *false* signalisiert werden.

Auf der anderen Seite sind Dienste vorstellbar, dessen Ausführung auch ohne Nutzer in Form anderer Komponenten sinnvoll ist. Hierzu zählen beispielsweise Dienste, welche eine grafische Benutzungsschnittstelle realisieren. Der Dienst, welcher von der Trainerkomponente bereitgestellt wird, ist hierfür ein typisches Beispiel. Er zeigt dem Trainer die Trainingsdaten seiner Sportler an und bietet u.a. die Möglichkeit zur Auswertung der Daten. Selbst, wenn keine andere Komponente diesen Dienst nutzen möchte, ist eine Ausführung sinnvoll. Derartige Dienste können dem Framework durch Setzen des Flags *requestRun* auf *true* kenntlich gemacht werden.

Wie später gezeigt wird, bilden Dienste mit gesetztem *requestRun*-Flag den Ausgangspunkt des Konfigurationsprozesses. Ziel des Konfigurationsprozesses wird es sein, diese Dienste auszuführen und hierzu, falls notwendig, Abhängigkeiten zu Diensten anderer Komponenten aufzulösen.

In der grafischen Darstellung werden Dienste, für die das Flag *requestRun* nicht gesetzt ist, durch einen leeren Kreis symbolisiert, während Dienste mit gesetztem *requestRun*-Flag mit Hilfe eines ausgefüllten Kreises dargestellt werden.

Die Klasse *ProvidedService* aus dem Modell in Abbildung 4-5 definiert ein drittes Attribut namens *maxNoOfUsers* in welchem eine ganze Zahl hinterlegt werden kann (siehe Formel 4-13):

$$\boldsymbol{maxNoOfUsers}: ProvidedService \rightarrow \mathbb{N}$$

Formel 4-13

Der Hintergrund ist der, dass Dienste häufig nicht von beliebig vielen Nutzern gleichzeitig verwendet werden können. Im Attribut *maxNoOfUsers* kann deshalb eine Obergrenze an Nutzern für den Dienst festlegt werden. Während des Konfigurationsprozesses stellt das Framework sicher, dass ein Dienst nur maximal von dieser Anzahl Nutzern gleichzeitig verwendet wird.

Neben den drei Attributen definiert die Klasse *ProvidedService* noch sieben Methoden. Deren Bedeutung wird in Abschnitt 4.6.3 näher erläutert. An dieser Stelle wird lediglich ein kurzer Überblick gegeben.

Über die Methode *notifyStateChanged* informiert das Framework die Komponente über Zustandswechsel des Automaten. Es handelt sich hierbei um eine abstrakte Methode, welche durch eine Komponente überschrieben werden kann, um so ggf. auf bestimmte Zustandswechsel reagieren zu können.

Bei Aufruf der Methode *install* durch die *DynamicAdaptiveComponent* wird der Dienst in den Ausgangszustand NOT_RUNNING versetzt, während er bei Aufruf von *uninstall* in den Endzustand übergeht und ggf. bestehende Verbindungen zu Nutzern beendet. Diese Methoden werden durch die *DynamicAdaptiveComponent* dann aufgerufen, wenn dort die Methode *install* bzw. *uninstall* aufgerufen wurde.

Über die Methode *serviceRunnable* wird ein Dienst durch die zughörige *DynamicAdaptiveComponent* darüber informiert, dass er ausgeführt werden kann. Dies ist immer dann der Fall, wenn ein Dienst mit einer aktiven *ComponentConfiguration* verbunden ist. Im Gegensatz dazu wird einem Dienst durch Aufruf des Trigger *serviceNotRunnable* mitgeteilt, dass er nicht mehr mit einer aktiven Komponentenkonfiguration verbunden ist. In diesem Fall wird die Ausführung des Dienstes gestoppt.

Die Methoden *wantsUse* und *wantsNotUse* nehmen eine Sonderstellung ein. Diese Methoden werden nämlich nicht durch Elemente der eigenen Komponente aufgerufen, sondern von *RequiredServiceReferenceSets* anderer Komponenten. Diese teilen dem Dienst durch Aufruf der Methode *wantsUse* mit, dass sie den Dienst gerne verwenden möchten. Daraufhin versucht dann das Framework die Abhängigkeiten für diesen Dienst aufzulösen. Hierzu informiert der Dienst alle verbundenen *ComponentConfigurations* durch Aufruf der Methode *mustRun* darüber, dass

ein von ihr angebotener Dienst ausgeführt werden möchte. Diese informiert daraufhin wiederum die verbundenen *RequiredServiceReferenceSets* durch Aufruf von *mustResolve* darüber, dass mit der Auflösung von Abhängigkeiten begonnen werden soll. Außerdem wird dieser Dienst der Menge *runRequestedBy* hinzugefügt (siehe Formel 4-9).

Auf der anderen Seite wird der Dienst durch Aufruf von *wantsNotUse* darüber informiert, dass ein Nutzer oder potentieller Nutzer weggefallen ist. Auch diese Information wird durch Aufruf von *mustNotRun* an die verbundenen *ComponentConfigurations* weitergegeben, welche daraufhin die Menge *runRequestedBy* aktualisieren. Ist die Menge *runRequestedBy* für eine *ComponentConfiguration* leer, so werden die verbundenen *RequiredServiceReferenceSets* durch Aufruf von *mustNotResolve* hierüber informiert. Details zu diesen Abläufen werden im Rahmen der Vorstellung der Zustandsautomaten in Abschnitt 4.6.3 erläutert. Zusätzlich wird in Abschnitt 4.8 ein Überblick über die Funktionsweise des Konfigurationsprozesses gegeben.

4.6.2 Beziehungen zu anderen Klassen des Komponentenmodells

Eine Komponente kann beliebig viele Dienste definieren und über eine oder mehrere *ComponentConfigurations* anbieten. Umgekehrt kann ein Dienst von beliebig vielen Komponentenkonfigurationen einer Komponente angeboten werden. Wie bereits zuvor erläutert wurde, müssen angebotene Dienste die *ComponentConfigurations* informieren, falls sie ausgeführt werden müssen oder nicht (durch Aufruf von *mustRun* bzw. *mustNotRun*). Hierzu benötigt jeder *ProvidedService* jeweils eine Referenz auf jede *ComponentConfiguration*, von denen der Dienst angeboten wird. Diese Menge ist im Modell durch die Assoziation *providedBy* repräsentiert (siehe Formel 4-14):

$$\boldsymbol{providedBy}: ProvidedService \rightarrow \mathcal{P}(ComponentConfiguration)$$

Formel 4-14

Wie in Abschnitt 4.2 bereits erläutert wurde, implementiert jeder Dienst genau eine Domänenschnittstelle. Die implementierte Schnittstelle ist im Modell durch die Assoziation *implements* beschrieben (siehe Formel 4-15):

$$\boldsymbol{implements}: ProvidedService \rightarrow DomainInterface$$

Formel 4-15

Neben den Assoziationen *providedBy* und *implements* besitzt die Klasse *ProvidedService* noch zwei Assoziationen zur Klasse *RequiredServiceReferenceSet*, nämlich *runRequestedBy* und *assignedTo*. Hierüber werden Beziehungen zwischen Komponenten abgebildet. Diese werden nun im Folgenden näher vorgestellt.

Wie bereits zuvor beschrieben gibt es für einen *ProvidedService* zwei Gründe, ausgeführt zu werden: das Flag *requestRun* ist gesetzt oder es existiert ein Nutzer bzw. Nutzungsinteressent in Form einer oder mehrere anderer Komponenten. Hier wird nun der zweite Fall näher betrachtet.

Ein *ProvidedService* kann durch andere Komponenten über deren *RequiredServiceReferenceSets* verwendet werden. Diese melden hierzu zunächst durch Aufruf von *wantsUse* beim Dienst den Nutzungswunsch an. Die für einen Dienst vorliegenden Nutzungsanfragen sind im Modell über die Assoziation *runRequestedBy* abgebildet (siehe Formel 4-16):

$$\boldsymbol{runRequestedBy}: ProvidedService \rightarrow \mathcal{P}(RequiredServiceReferenceSet)$$

Formel 4-16

Wie im folgenden Abschnitt noch erläutert wird, kann eine Nutzungsanfrage jederzeit gestellt werden, egal ob sich ein Dienst im Zustand NOT_RUNNING, RUNNABLE oder RUNNING befindet. Verwendet werden kann er allerdings erst dann, wenn der Dienst dem potentiellen Dienstnutzer die Nutzungsrechte durch Aufruf des Triggers *serviceAssigned* zuweist. Befindet sich ein Dienst zum Zeitpunkt der Nutzungsanfrage im Zustand RUNNING und ist die maximale Anzahl an Dienstnutzern (*maxNoOfUsers*) noch nicht erreicht, so erfolgt unmittelbar die Zuordnung des Dienstes an den Nutzer. Andernfalls wird die Zuordnung solange verzögert, bis sich der Dienst im Zustand RUNNING befindet und die maximale Zahl von Dienstnutzern nicht erreicht ist.

Eine Konkurrenzsituation um die Nutzungsrechte eines Dienstes entsteht immer dann, wenn die Anzahl der Elemente in *runRequestedBy* größer ist als in *maxNoOfUsers* hinterlegte Zahl. In diesem Fall weist das Framework die Dienste in der Reihenfolge des Eingangs von Nutzungsanfragen zu.

Jeder *ProvidedService* verwaltet neben einer Liste der Nutzungsanfragen eine Liste mit denjenigen *RequiredServiceReferenceSets*, denen die Nutzungsrechte bereits zugeteilt wurden. Diese ist im Modell durch die Assoziation *assignedTo* repräsentiert (siehe Formel 4-17):

$$\boldsymbol{assignedTo}: ProvidedService \rightarrow \mathcal{P}(RequiredServiceReferenceSet)$$
$$\forall ps \in ProvidedService:$$
$$assignedTo(ps) \subseteq runRequestedBy(ps) \wedge$$
$$|assignedTo(ps)| \leq maxNoOfUsers(ps)$$

Formel 4-17

Diese Menge wird später u.a. benötigt, um im Falle einer Beendigung des Dienstes oder der gesamten Komponente die Nutzer informieren zu können, dass eine weitere Nutzung des Dienstes nicht möglich ist.

4.6.3 Der Zustandsautomat von *ProvidedService*

Der Zustandsautomat eines ProvidedServices besteht aus drei Zuständen, nämlich NOT_RUNNING, RUNNABLE und RUNNING. Abbildung 4-13 zeigt den Zustandsautomaten inklusive aller Zustandsübergänge.

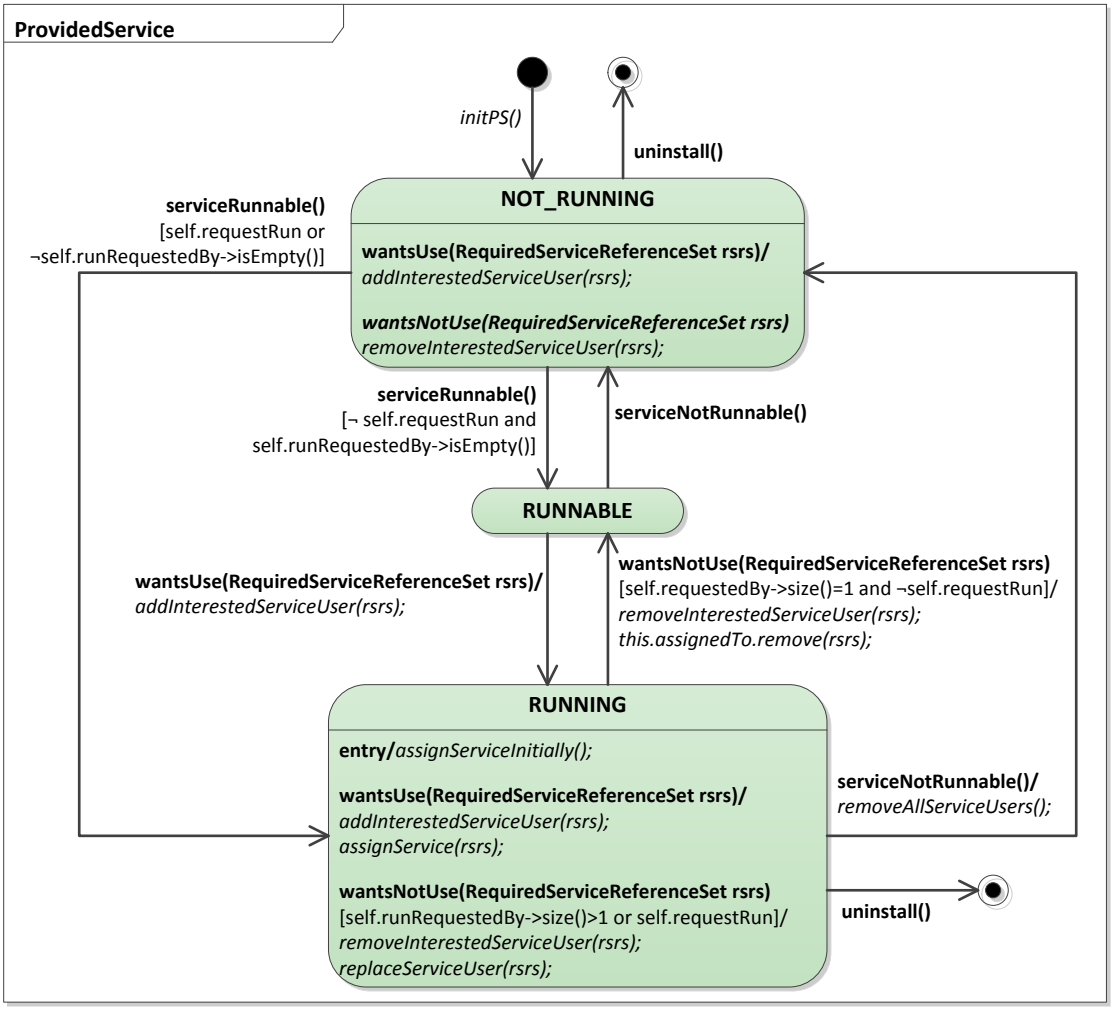


Abbildung 4-13: Der Zustandsautomat von *ProvidedService*

Nach Installation eines *ProvidedService* wird er zunächst allen potentiellen Nutzern im System bekannt gemacht. Falls für diesen Dienst das Flag *requestRun* gesetzt wurde, so wird außerdem allen verbundenen *ComponentConfigurations* mitgeteilt, dass ein Dienst existiert, welcher ausgeführt werden möchte. Diese Funktionalität ist im Zustandsautomaten durch die Funktion *initPS* realisiert, welche in Listing 4-4 abgebildet ist.

```
1  initPS()  {
2   forall rsrs in RequiredServiceReferenceSet {
3    rsrs.newService(this);
4   }
5   if(this.requestRun) {
6    for(ComponentConfiguration c : this.providedBy) {c.mustRun(this);}
7   }
8  }
```

Listing 4-4: Die Methode *initPS*

Anschließend befindet sich ein Dienst in seinem Startzustand NOT_RUNNING, welcher im folgenden Abschnitt vorgestellt wird.

Der Zustand NOT_RUNNING

Der Zustand NOT_RUNNING ist der Startzustand eines jeden angebotenen Dienstes. In diesem Zustand wird der Dienst nicht ausgeführt und kann somit nicht verwendet werden, weder von anderen Komponenten, noch von Endanwendern.

Ein Dienst geht dann in den Zustand RUNNING oder RUNNABLE über, sobald er von der Komponente durch Aufruf von *serviceRunnable* mitgeteilt bekommt, dass er ausgeführt werden kann. Dies geschieht immer dann, wenn ein Dienst mit einer aktiven *ComponentConfiguration* verknüpft ist. Solange dies nicht der Fall ist, reagiert der Dienst lediglich auf Nutzungsanfragen anderer Komponenten. Denn diese Anfragen muss jeder Dienst aus zwei Gründen verwalten. Zum einen müssen verbundene *ComponentConfigurations* darüber informiert werden, dass sie nun einen angebotenen Dienst besitzen, welcher ausgeführt werden möchte. Die Komponentenkonfigurationen beginnen daraufhin mit der Auflösung von Abhängigkeiten zu Diensten anderer Komponenten. Zum anderen wählt der Dienst später aus der Menge der interessierten Nutzer diejenigen aus, welche den Dienst verwenden dürfen. Alle interessierten Nutzer eines Dienstes werden im Modell durch die bereits vorgestellte Assoziation *runRequestedBy* repräsentiert. Hierbei handelt es sich um eine geordnete Liste ohne Duplikate, welche von jedem *ProvidedService* verwaltet wird. Bei Aufruf des Triggers *wantsUse* wird diese Menge entsprechend erweitert und wie bereits erwähnt sämtliche *ComponentConfigurations* durch Aufruf des Triggers *mustRun* informiert. Die Methode *newInterestedServiceUser* des Zustandsautomaten realisiert dieses Verhalten und ist in Listing 4-5 beschrieben:

```
1  addInterestedServiceUser(RequiredServiceReferenceSet rsrs)  {
2   this.runRequestedBy.add(rsrs);
3   for(ComponentConfiguration c : this.providedBy) {c.mustRun(this);}
4  }
```

Listing 4-5: Die Methode *addInterestedServiceUser*

Ebenso muss ein angebotener Dienst reagieren, falls ein potentieller Nutzer ausfällt. Dies teilen diese Nutzer den Diensten durch Aufruf des Triggers *wantsNotUse* mit. Sind keine potentiellen Nutzer mehr vorhanden und das Flag *requestRun* nicht gesetzt, so muss dieser Dienst nicht mehr ausgeführt werden. In diesem Fall muss das den verbundenen *ComponentConfigurations* durch Aufruf des Triggers *mustNotRun* mitgeteilt werden. Das Listing 4-6 fasst das beschriebene Verhalten zusammen.

```
1  removeInterestedServiceUser(RequiredServiceReferenceSet rsrs)  {
2   this.runRequestedBy.remove(rsrs);
3   if(this.runRequestedBy.isEmpty() && !this.requestRun) {
4    for(ComponentConfiguration c : this.providedBy) {c.mustNotRun(this);}
5   }
6  }
```

Listing 4-6: Die Methode remove*InterestedServiceUser*

Ausgeführt werden kann ein Dienst nur dann, wenn er von der Komponente über den Trigger *serviceRunnable* mitgeteilt bekommt, dass alle Abhängigkeiten aufgelöst werden konnten. Gibt es nun für diesen Dienst bei Aufruf des Triggers keine potentiellen Nutzer und ist zudem das Flag *requestRun* nicht gesetzt, so geht der Automat in den Zustand RUNNABLE über. Er wird dann also deshalb nicht ausgeführt, weil es für ihn keine Verwendung gibt. Andernfalls wird der Dienst ausgeführt und wechselt in den Zustand RUNNING.

Der Zustand RUNNABLE

Befindet sich ein Dienst im Zustand RUNNABLE, so ist er mit einer aktiven *ComponentConfiguration* verbunden. Das bedeutet, alle Abhängigkeiten für diesen Dienst konnten aufgelöst werden und der Dienst ist bereit zur Ausführung. Solange allerdings keine Nutzungsanfrage durch eine andere Komponente vorliegt, wird der Dienst nicht gestartet. Erst bei Aufruf des Triggers *wantsUse* durch ein *RequiredServiceReferenceSet* einer anderen Komponente wechselt der Dienst in den Zustand RUNNING. Hierbei wird die Methode *addInterestedServiceUser* abgearbeitet, welche bereits in Listing 4-5 vorgestellt wurde.

Bekommt der Dienst über den Trigger *serviceNotRunnable* durch die Komponente mitgeteilt, dass der Dienst nicht mehr ausgeführt werden kann, so wechselt der Dienst zurück in den Zustand NOT_RUNNING. Da im Zustand RUNNABLE keine Nutzungsanfragen vorliegen und die Menge *runRequestedBy* somit leer ist, müssen in diesem Fall keine weiteren Aktionen durchgeführt werden.

Der Zustand RUNNING

Befindet sich ein Dienst im Zustand RUNNING, so wird der Dienst ausgeführt und er kann verwendet werden. Bei Eintritt in den Zustand entscheidet der Dienst zunächst, welcher der in *runRequestedBy* enthaltenen *RequiredServiceReferenceSets* den Dienst verwenden dürfen. Eine Auswahl muss dann getroffen werden, wenn die Anzahl der Elemente in *runRequestedBy* die im Attribut *maxNoOfUsers* hinterlegte Zahl überschreitet. Das Framework teilt in diesem Fall die Nutzungsrechte gemäß dem Eingang der Nutzungsanfragen zu. Die Dienstzuteilung geschieht durch Aufruf von *serviceAssigned* bei den betreffenden *RequiredServiceReferenceSets.* Diese werden außerdem in die Menge *assignedTo* aufgenommen. Das Listing 4-7 fasst die Aktionen zusammen.

```
1  assignServiceInitially()  {
2   for(RequiredServiceReferenceSet rsrs :
3                          this.runRequestedBy.subset(0, maxNoOfUsers)) {
4    rsrs.serviceAssigned(this);
5    this.assignedTo.add(rsrs);
6   }
7  }
```

Listing 4-7: Die Methode *assignServiceInitially*

Während sich ein Dienst im Zustand RUNNING befindet, werden weiterhin Nutzungsanfragen entgegengenommen. Solange ein Dienst weniger Nutzer hat als im Attribut *maxNoOfUsers* angegeben, kann dieser Dienst im Falle einer Nutzungsanfrage unmittelbar verwendet werden. Aber auch wenn die Obergrenze an Nutzern erreicht ist, werden weiterhin Anfragen entgegengenommen. Fällt ein Nutzer aus, kann einem anderen potentiellen Nutzer der Dienst zugeordnet werden. Wird nun also eine Nutzungsanfrage in Form des Triggers *wantsUse* gestellt, so wird zunächst die aus Listing 4-5 bekannte Methode *addInterestedServiceUser* aufgerufen und anschließend die in Listing 4-8 definierte Methode *assignService*.

```
1  assignService(RequiredServiceReferenceSet rsrs)  {
2   if(this.assignedTo.size()<this.maxNoOfUsers) {
3    rsrs.serviceAssigned(this);
4    this.assignedTo.add(rsrs);
5   }
6  }
```

Listing 4-8: Die Methode *assignService*

Wie gerade bereits angedeutet wurde, kann es vorkommen, dass ein Nutzer ausfällt und den Dienst dementsprechend nicht mehr benötigt. Dies wird dem Dienst über den Trigger *wantsNotUse* mitgeteilt. Im Zustand RUNNING müssen hier zwei Situationen unterschieden werden. Handelte es sich bei dem weggefallenen Nutzer um den letzten Nutzer des Dienstes und ist gleichzeitig das Flag *requestRun* nicht gesetzt, so besteht keine Notwendigkeit mehr, den Dienst auszuführen. Die verbundenen *ComponentConfigurations* werden hierüber informiert (Listing 4-6) und der Dienst wechselt in den Zustand RUNNABLE. War die Kapazität des Dienstes vor Wegfall des Nutzers ausgeschöpft und existieren weitere potentielle Nutzer in der Menge *runRequestedBy*, so kann der Dienst einem *RequiredServiceReferenceSet* dieser Menge zugeordnet werden. Die Methode *replaceServiceUser* aus Listing 4-9 realisiert diese Funktionalität.

```
1  replaceServiceUser(RequiredServiceReferenceSet oldUser)  {
2   this.assignedTo.remove(oldUser);
3   for(RequiredServiceReferenceSet rsrs : this.runRequestedBy) {
4    if(!this.assignedTo.contains(rsrs)) {
5     rsrs.serviceAssigned(this);
6     this.assignedTo.add(rsrs);
7     return;
8    }
9   }
10 }
```

Listing 4-9: Die Methode *replaceServiceUser*

Wird dem Dienst durch die Komponente über den Trigger *serviceNotRunnable* mitgeteilt, dass der Dienst nicht mehr ausgeführt werden kann, so wechselt er zurück in den Zustand NOT_RUNNING. Hierbei müssen alle Nutzer darüber informiert werden, dass der Dienst nun nicht mehr verwendet werden kann. Dies geschieht durch Aufruf des Triggers *serviceNotAssigned* bei allen *RequiredServiceReferenceSets* der Menge *assignedTo*. Die Methode *removeAllServiceUsers* aus Listing 4-10 realisiert diese Funktionalität.

```
1  removeAllServiceUsers()  {
2   for(RequiredServiceReferenceSet rsrs : this.assignedTo) {
3    rsrs.serviceNotAssigned(this);
4   }
5   this.assignedTo.clear();
6  }
```

Listing 4-10: Die Methode *removeAllServiceUsers*

Das Zusammenspiel der einzelnen Elemente einer Komponente sowie zwischen Komponenten wird in Abschnitt 4.8 anhand eines Beispiels nochmals erläutert.

4.7 Spezifikation benötigter Dienste

Angebotene Dienste einer Komponente benötigen häufig Zugriff auf Dienste anderer Komponenten um ausgeführt werden zu können. Erst wenn derartige Abhängigkeiten aufgelöst sind, kann auch der betreffende Dienst gestartet werden. Derartige Abhängigkeiten zu Diensten anderer Komponenten werden im Framework über die Klasse *RequiredServiceReferenceSet* definiert.

Ein *RequiredServiceReferenceSet* definiert Abhängigkeiten zu Diensten anderer Komponenten. Hierbei definiert jedes *RequiredServiceReferenceSet* eine Abhängigkeit zu Diensten, welche eine

bestimmte Domänenschnittstelle implementieren. So können für eine Komponente beispielsweise zwei *RequiredServiceReferenceSets* definiert werden, wobei eines Abhängigkeiten zu Diensten definiert, welche eine Schnittstelle I_1 implementieren, und ein zweites welche Abhängigkeiten zu Diensten definiert, welche eine Schnittstelle I_2 implementieren. Der Aufbau und die Funktionsweise von *RequiredServiceReferenceSets* wird im Folgenden detailliert vorgestellt.

4.7.1 Methoden und Attribute der Klasse *RequiredServiceReferenceSet*

Die Klasse *RequiredServiceReferenceSet* definiert drei Attribute sowie neun Methoden und ist in Abbildung 4-14 nochmals dargestellt.

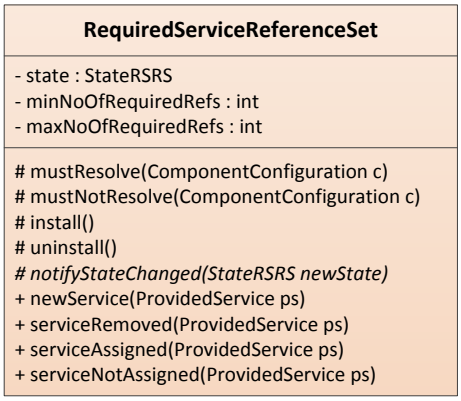


Abbildung 4-14: Die Klasse RequiredServiceReferenceSet

Genau wie alle zuvor vorgestellten Elemente einer Komponente realisiert auch die Klasse *RequiredServiceReferenceSet* einen Zustandsautomaten, dessen aktueller Zustand im Attribut *state* hinterlegt ist (siehe Formel 4-18):

$$\boldsymbol{StateRSRS} = \{NOT\_RESOLVED, RESOLVING, RESOLVED\}$$
$$\boldsymbol{state} \in StateRSRS$$

Formel 4-18

Nach Installation einer Komponente befinden sich dessen *RequiredServiceReferenceSets* zunächst im Zustand NOT_RESOLVED. In diesem Zustand gelten die Abhängigkeiten, welche durch das *RequiredServiceReferenceSet* definiert sind, als nicht aufgelöst. Außerdem versucht das Framework in diesem Zustand nicht, die Abhängigkeiten aufzulösen. Im Zustand RESOLVING hingegen wird versucht, geeignete Dienste anderer Komponente zu finden und Nutzungsrechte für diese zu erlangen. Für *RequiredServiceReferenceSets* im Zustand RESOLVED ist dies gelungen. Dies wird dann den verbundenen *ComponentConfigurations* durch Aufruf von *rsrsResolved* mitgeteilt. Dies kann wiederum dazu führen, dass Dienste der Komponente ausführbar werden und wiederum anderen Komponenten zur Verfügung gestellt werden können. Eine detaillierte Vorstellung des zugehörigen Zustandsautomaten wird in Abschnitt 4.7.3 gegeben.

Neben dem Attribut *state* definiert die Klasse *RequiredServiceReferenceSet* noch zwei weitere Attribute, nämlich *minNoOfRequiredRefs* und *maxNoOfRequiredRefs*, welche jeweils eine positive ganze Zahl speichern (siehe Formel 4-19):

$$\boldsymbol{minNoOfRequiredRefs}: RequiredServiceReferenceSet \rightarrow \mathbb{N}_0$$
$$\boldsymbol{maxNoOfRequiredRefs}: RequiredServiceReferenceSet \rightarrow \mathbb{N} \cup \infty$$
$$\forall rsrs \in RequiredServiceReferenceSet:$$
$$maxNoOfRequiredRefs(rsrs) \geq minNoOfRequiredRefs(rsrs)$$

Formel 4-19

Wie bereits erläutert, definiert jedes *RequiredServiceReferenceSet* eine Abhängigkeit zu Diensten anderer Komponenten. Im hier beschriebenen Komponentenmodell gilt die Abhängigkeit eines *RequiredServiceReferenceSet* dann als aufgelöst, falls mindestens so viele Dienste gefunden und genutzt werden können, wie im Attribut *minNoOfRequiredRefs* angegeben ist. Hierbei müssen diese Dienste eine bestimmte Domänenschnittstelle implementieren, welche durch den Komponentenentwickler vorgegeben werden muss. Abbildung 4-15 zeigt beispielsweise eine Komponente mit zwei *RequiredServiceReferenceSets*, welche durch eine *ComponentConfiguration* definiert werden.

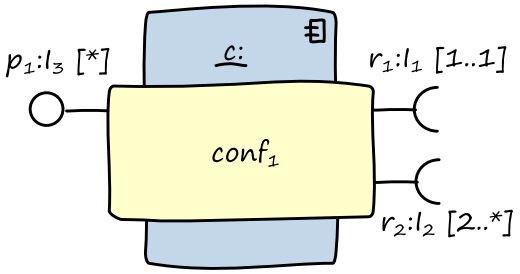


Abbildung 4-15: Beispiel einer Komponente mit zwei *RequiredServiceReferenceSets*

In diesem Fall kann der *ProvidedService* p_1 nur dann ausgeführt werden, wenn sowohl die Abhängigkeiten des *RequiredServiceReferenceSets* r_1 als auch die von r_2 aufgelöst sind. Die durch r_1 definierte Abhängigkeit ist dann aufgelöst, wenn ein Dienst gefunden wurde, welcher die Schnittstelle I_1 implementiert. Und die Abhängigkeit, welche durch r_2 definiert ist, gilt dann als aufgelöst, falls mindestens zwei Dienste gefunden wurden, die die Domänenschnittstelle I_2 implementieren. Wie später noch gezeigt wird, reicht es hierbei jedoch nicht aus, dass die Dienste verfügbar sind. Sie müssen sich zusätzlich im Zustand RUNNING befinden und es müssen den jeweiligen *RequiredServiceReferenceSets* die Nutzungsrechte durch die Dienste erteilt worden sein.

Sind mehr Dienste für ein *RequiredServiceReferenceSet* verfügbar als benötigt, so muss eine Auswahl der zu nutzenden Dienste durch das Framework getroffen werden. Die maximale Anzahl benötigter Referenzen ist im Attribut *maxNoOfRequiredRefs* angegeben (siehe Formel 4-19). Für das *RequiredServiceReferenceSet* r_1 wird beispielsweise maximal ein Dienst benötigt, der die Domänenschnittstelle I_1 implementiert. Der hier gewählte Ansatz zur Auswahl eines Dienstes sieht vor, denjenigen Dienst zu verwenden, welcher dem *RequiredServiceReferenceSet* zuerst die Nutzungsrechte erteilt hat. Sobald die obere Grenze an benötigten Diensten erreicht ist, ziehen die betreffenden *RequiredServiceReferenceSets* ggf. bestehende Nutzungsanfragen bei den übrigen Diensten zurück.

Mit Hilfe von *RequiredServiceReferenceSets* ist es somit möglich, Anforderungen hinsichtlich der Anzahl benötigter Dienste zu spezifizieren, als auch hinsichtlich der implementierten

Domänenschnittstelle. In Kapitel 5 werden die Möglichkeiten zur Spezifikation von Anforderungen noch dahingehend erweitert, dass zusätzlich Anforderungen an die Rückgabewerte von Methoden der jeweils referenzierten Domänenschnittstelle spezifiziert werden können.

4.7.2 Beziehungen zu anderen Klassen des Komponentenmodells

Jedes *RequiredServiceReferenceSet* verwaltet eine Menge von Referenzen auf Dienste anderer Komponenten. Wie bereits beschrieben, spezifiziert jedes *RequiredServiceReferenceSet* einer Komponente Abhängigkeiten zu einer Menge von Diensten anderer Komponenten, welche eine bestimmte Domänenschnittstelle implementieren. So könnte beispielsweise eine Trainerkomponente ein *RequiredServiceReferenceSet* definieren, welche alle Referenzen auf solche benötigten Dienste verwaltet, die die Domänenschnittstelle *IAthlete* implementieren. Welchen Typ von Diensten ein *RequiredServiceReferenceSet* verwaltet, wird über die Assoziation *refersTo* im Modell aus Abbildung 4-5 definiert (siehe Formel 4-20):

$$\boldsymbol{refersTo}: RequiredServiceReferenceSet \rightarrow DomainInterface$$

Formel 4-20

Eine Komponente kann beliebig viele *RequiredServiceReferenceSets* definieren. *ComponentConfigurations* wiederum können mit beliebig vielen dieser *RequiredServiceReferenceSets* verbunden werden (siehe Formel 4-8). Und gleichzeitig kann jedes *RequiredServiceReferenceSet* mit beliebig vielen *ComponentConfigurations* einer Komponente verbunden sein. Letztere Verbindung ist im Modell durch die Assoziation *declaredBy* realisiert (siehe Formel 4-21):

$$\boldsymbol{declaredBy}: RequiredServiceReferenceSet \rightarrow \mathcal{P}(ComponentConfiguration)$$

Formel 4-21

Die Abhängigkeiten, welche durch ein *RequiredServiceReferenceSet* spezifiziert sind, werden durch das Framework nicht unmittelbar nach Instanziierung oder Installation aufgelöst. Vielmehr wird mit diesem Prozess erst dann begonnen, wenn sie durch Aufruf der Methode *mustResolve* durch eine *ComponentConfiguration* dazu aufgefordert wurde. Erst daraufhin stellt dann ein *RequiredServiceReferenceSet* eine Nutzungsanfrage an alle verfügbaren Dienste, welche dieselbe Domänenschnittstelle implementieren wie die, welche vom jeweiligen *RequiredServiceReferenceSet* referenziert wird. Wie bereits erwähnt, wird diese Nutzungsanfrage durch Aufruf von *wantsUse* bei den betreffenden Diensten gestellt. Die Menge derjenigen *ComponentConfigurations*, welche eine Aufforderung zur Auflösung der Abhängigkeiten übermittelt hat, ist im Modell durch die Assoziation *resolveRequestedBy* repräsentiert (siehe Formel 4-22):

$$\boldsymbol{resolveRequestedBy}: RequiredServiceReferenceSet \rightarrow \mathcal{P}(ComponentConfiguration)$$

Formel 4-22

Sobald diese Menge also nicht leer ist, wird mit der Auflösung von Abhängigkeiten für das jeweilige *RequiredServiceReferenceSet* begonnen. *ComponentConfigurations* können Aufforderungen zur Auflösung von Abhängigkeiten durch Aufruf von *mustNotResolve* auch wieder zurücknehmen. Sobald durch keine *ComponentConfiguration* mehr eine Auflösung von

Abhängigkeiten gewünscht ist und die Menge *resolveRequestedBy* somit leer ist, werden alle bereits belegten Dienste durch Aufruf von *wantsNotUse* freigegeben und die Suche nach geeigneten Diensten wird eingestellt.

Damit die durch ein *RequiredServiceReferenceSet* definierten Abhängigkeiten aufgelöst werden können, muss für jedes die Menge der zur Verwendung in Frage kommenden Dienste bestimmt werden. Diese Menge der potentiell nutzbaren, durch andere Komponenten angebotenen Dienste, wird im Modell aus Abbildung 4-5 durch die Assoziation *canUse* repräsentiert. (siehe Formel 4-23):

$$\boldsymbol{canUse}: RequiredServiceReferenceSet \rightarrow \mathcal{P}(ProvidedService)$$
$$\forall rsrs \in RequiredServiceReferenceSet, \forall ps \in canUse(rsrs): implements(ps) = refersTo(rsrs)$$

Formel 4-23

Diese Menge enthält diejenigen *ProvidedServices*, welche genau die Domänenschnittstelle implementieren, die vom *RequiredServiceReferenceSet* über *refersTo* referenziert wird. Sie wird durch das Framework jederzeit aktuell gehalten, um im Falle der Notwendigkeit, Abhängigkeiten aufzulösen, unmittelbar entsprechende Nutzungsanfragen stellen zu können. Kommen also neue kompatible Dienste im System hinzu bzw. verlassen das System, so wird die Menge durch das Framework entsprechend angepasst.

Bekommt nun ein *RequiredServiceReferenceSet* durch Aufruf der Methode *mustResolve* von einer *ComponentConfiguration* mitgeteilt, dass mit dem Auflösen von Abhängigkeiten begonnen werden soll, so stellt das *RequiredServiceReferenceSet* eine Nutzungsanfrage an alle *ProvidedServices* der Menge *canUse*. Die Menge derjenigen Dienste, für die eine Nutzungsanfrage gestellt wurde, ist im Modell durch die Assoziation *wantsUse* repräsentiert (siehe Formel 4-24):

$$\boldsymbol{wantsUse}: RequiredServiceReferenceSet \rightarrow \mathcal{P}(ProvidedService)$$
$$\forall rsrs \in RequiredServiceReferenceSet: wantsUse(rsrs) \subseteq canUse(rsrs)$$

Formel 4-24

Wie bereits im vorherigen Abschnitt erläutert wurde, kann ein *RequiredServiceReferenceSet* angebotene Dienste anderer Komponenten erst dann nutzen, wenn ihm die Nutzungsrechte durch Aufruf von *serviceAssigned* zugeordnet worden sind. Diejenigen Dienste, welche dieses Nutzungsrecht erteilt haben, werden im Komponentenmodell durch die Assoziation *uses* repräsentiert (siehe Formel 4-25):

$$\boldsymbol{uses}: RequiredServiceReferenceSet \rightarrow \mathcal{P}(ProvidedService)$$
$$\forall rsrs \in RequiredServiceReferenceSet: uses(rsrs) \subseteq wantsUse(rsrs) \wedge minNoOfUsers(rsrs) \leq |uses(rsrs)| \leq maxNoOfUsers(rsrs)$$

Formel 4-25

Die Abhängigkeiten, welche durch ein *RequiredServiceReferenceSet* definiert sind, gelten dann als aufgelöst, falls die Menge *uses* mindestens die im Attribut *minNoOfRequiredRefs* (siehe Formel 4-19) spezifizierte Anzahl an *ProvidedServices* enthält. Des Weiteren sorgt das Framework dafür, dass die Menge *uses* nicht mehr Elemente enthält, als im Attribut *maxNoOfRequiredRefs* angegeben ist.

Sämtliche Abläufe werden sowohl im nun folgenden Abschnitt etwas näher erläutert, als auch im Abschnitt 4.8 nochmals im Überblick dargestellt.

4.7.3 Der Zustandsautomat von RequiredServiceReferenceSet

Genau wie alle anderen Elemente einer Komponente, realisiert auch ein *RequiredServiceReferenceSet* einen Zustandsautomaten. Dieser besteht aus den bereits in Formel 4-18 angegebenen Zuständen NOT_RESOLVED, RESOLVING und RESOLVED. Abbildung 4-16 zeigt den zugehörigen Zustandsautomaten. Im Folgenden werden nun die einzelnen Zustände sowie die Zustandsübergänge im Detail erläutert.

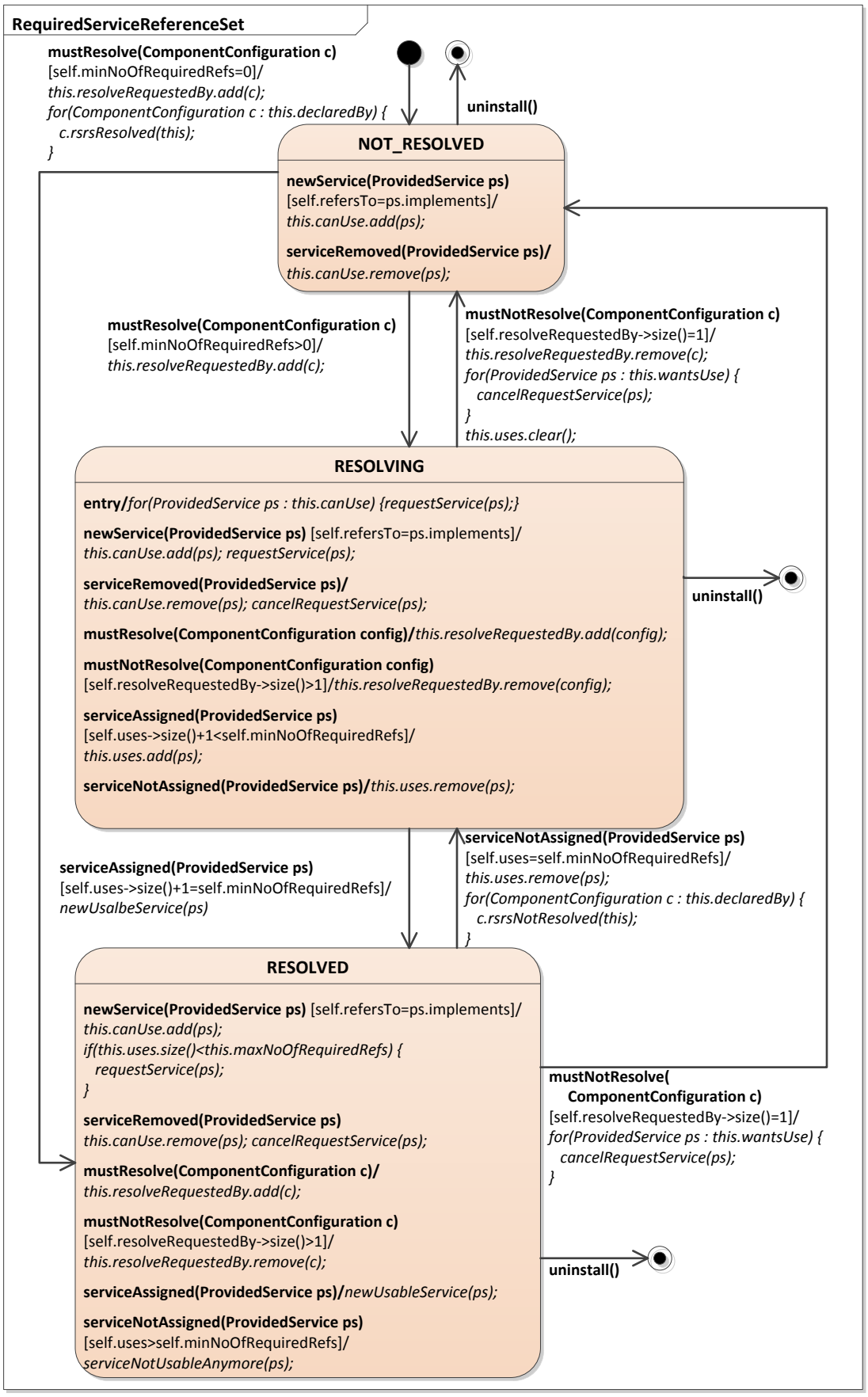


Abbildung 4-16: Der Zustandsautomat von *RequiredServiceReferenceSet*

Der Zustand NOT_RESOLVED

Nachdem eine Komponente durch Aufruf von *install* installiert wurde, werden alle *RequiredServiceReferenceSets* zunächst in den Zustand NOT_RESOLVED versetzt. In diesem Zustand gelten die definierten Abhängigkeiten als nicht aufgelöst und das Framework versucht auch nicht, Dienste für eine Verwendung zu reservieren. Für jedes *RequiredServiceReferenceSet* werden lediglich die Menge der potentiell in Frage kommenden Dienste (*canUse*) stetig aktuell gehalten. Hierzu reagiert der Automat auf die Trigger *newService* und *serviceRemoved*. Die Methode *newService* wird vom Framework immer dann aufgerufen, wenn ein neuer Dienst im System verfügbar wird. Ist die durch den neuen Dienst implementierte Domänenschnittstelle identisch mit derjenigen, welche von einem *RequiredServiceReferenceSet* referenziert wird (*refersTo*), so wird er in die Menge *canUse* des jeweiligen *RequiredServiceReferenceSets* aufgenommen. Genauso wird ein *ProvidedService* aus der Menge *canUse* entfernt, für den Fall, dass er nicht mehr im System verfügbar ist.

Ein *RequiredServiceReferenceSet* wechselt erst dann in einen anderen Zustand, wenn ihm signalisiert wird, dass die Auflösung der definierten Abhängigkeiten erforderlich ist. Dies wird *RequiredServiceReferenceSets* von verbundenen *ComponentConfigurations* durch Aufruf des Triggers *mustResolve* signalisiert. Wie bereits erwähnt, werden diejenigen *ComponentConfigurations*, welche eine Anfrage zur Auflösung von Abhängigkeiten gestellt haben, in die Menge *resolveRequestedBy* aufgenommen. Auf diese Weise bestimmt das Framework, ob das Auflösen von Abhängigkeiten notwendig ist oder nicht. Im Zustand NOT_RESOLVED ist die Menge stets leer.

Das *RequiredServiceReferenceSet* wechselt daraufhin entweder direkt in den Zustand RESOLVED oder zunächst in den Zustand RESOLVING. Ersteres tritt dann ein, wenn die minimale Anzahl an benötigten Referenzen auf angebotene Dienste anderer Komponenten gleich Null ist, also das Attribut *minNoOfRequiredRefs* den Wert 0 enthält. Wird allerdings mindestens ein Dienst benötigt, so wechselt das *RequiredServiceReferenceSet* zunächst in den Zustand RESOLVING. Hier verbleibt er dann solange, bis er mindestens die im Attribut *minNoOfRequiredRefs* angegebene Anzahl an nutzbaren Diensten zur Verfügung hat.

Wird eine Komponente deinstalliert, so wird dies durch Aufruf des Triggers *uninstall* allen *RequiredServiceReferenceSets* der Komponente mitgeteilt. Daraufhin geht der Automat in den Endzustand über.

Der Zustand RESOLVING

Befindet sich ein *RequiredServiceReferenceSet* im Zustand RESOLVING, so versucht das Framework die Abhängigkeiten aufzulösen, welche das jeweilige *RequiredServiceReferenceSet* definiert. Hierbei verbleibt es solange in diesem Zustand bis entweder die Abhängigkeiten aufgelöst werden konnten, oder aber ein Auflösen nicht mehr erforderlich ist.

Bei Eintritt in den Zustand RESOLVING wird zunächst eine Nutzungsanfrage an alle Dienste der Menge *canUse* des *RequiredServiceReferenceSets* gestellt. Die Methode *requestService* aus Listing 4-11 realisiert diese Funktionalität.

```
1  requestService(ProvidedService ps)  {
2    this.wantsUse.add(ps);
3    ps.wantsUse(this);
4  }
```

Listing 4-11: Die Methode *requestService*

An dieser Stelle werden ggf. mehr Dienste angefragt, als später tatsächlich benötigt werden. Dies ist immer dann der Fall, wenn die Anzahl der *ProvidedServices* in der Menge *canUse* die Anzahl übersteigt, welche im Attribut *maxNoOfRequiredRefs* spezifiziert ist. Trotzdem wird eine Nutzungsanfrage zunächst an alle Dienste der Menge *canUse* gestellt, da zunächst nicht davon ausgegangen werden kann, dass auf eine Anfrage auch eine Dienstzuordnung erfolgt und damit die Möglichkeit zur Nutzung besteht (siehe Abschnitt 4.6).

Solange im Zustand RESOLVING neue Dienste hinzukommen, wird an diese jeweils eine Nutzungsanfrage gestellt. Steht ein bislang verwendeter Dienst nicht mehr zur Verfügung, so wird die Anfrage zurückgezogen und der Dienst aus der Menge *wantsUse* entfernt. Die Methode *cancelRequestService* aus Listing 4-12 realisiert dies.

```
1  cancelRequestService(ProvidedService ps)  {
2    this.wantsUse.remove(ps);
3    ps.wantsNotUse(this);
4  }
```

Listing 4-12: Die Methode *cancelRequestService*

Ein *RequiredServiceReferenceSet* verbleibt solange im Zustand RESOLVING, bis ihm mindestens so viele Dienste zugeteilt wurden, wie im Attribut *minNoOfRequiredRefs* angegeben ist, oder bis eine Auflösung der Abhängigkeiten nicht mehr nötig ist.

Die Zuteilung eines Dienstes zu einem *RequiredServiceReferenceSet* wird diesem über den Trigger *serviceAssigned* signalisiert. Der Dienst wird der Menge *uses* hinzugefügt, solange, bis genügend Dienste verfügbar sind. Sobald die Menge *uses* mindestens die im Attribut *minNoOfRequiredRefs* hinterlegte Anzahl an Diensten enthält, wechselt das *RequiredServiceReferenceSet* in den Zustand RESOLVED. Hat die Anzahl der *ProvidedServices*, die einem *RequiredServiceReferenceSet* zur Verwendung zur Verfügung stehen, die Zahl erreicht, die im Attribut *maxNoOfRequiredRefs* spezifiziert ist, werden alle übrigen Nutzungsanfragen durch Aufruf von *wantsNotUse* zurückgezogen. Die durchgeführten Aktionen beim Zustandsübergang von RESOLVING nach RESOLVED sind im Listing 4-13 zusammengefasst.

```
1  newUsableService(ProvidedService ps)  {
2   if(this.uses.size()<this.maxNoOfRequiredRefs) {
3    this.uses.add(ps);
4    if(this.uses.size()==this.maxNoOfRequiredRefs) {
5     for(ProvidedService p : this.wantsUse) {
6      if(!uses.contains(p)) {cancelRequestService(p);}
7     }
8    }
9   }
10 }
```

Listing 4-13: Die Methode *newUsableService*

Wird einem *RequiredServiceReferenceSet* ein Dienst entzogen, so wird ihm dies über den Trigger *serviceNotAssigned* mitgeteilt. Der Dienst wird daraufhin aus der Menge *uses* entfernt, wobei der Automat im Zustand RESOLVING verbleibt.

Erst, falls keine Notwendigkeit mehr für das Auflösen von Abhängigkeiten besteht, wechselt der Automat zurück in den Zustand NOT_RESOLVED. Dies ist dann der Fall, wenn die Menge *resolveRequestedBy* leer ist, also keine *ComponentConfiguration* mehr existiert, für die ein Auflösen der Abhängigkeiten notwendig ist. Bei jedem Aufruf des Triggers *mustNotResolve* durch eine *ComponentConfiguration* wird deshalb überprüft, ob es sich hierbei um die letzte Komponentenkonfiguration mit Bedarf an Diensten anderer Komponenten handelt.

Der Zustand RESOLVED

Im Zustand RESOLVED sind die Abhängigkeiten zu benötigten Diensten aufgelöst. Das heißt, dass die Menge *uses* mindestens die Anzahl an Referenzen auf Dienste anderer Komponenten enthält, welche im Attribut *minNoOfRequiredRefs* spezifiziert wurde.

Im Zustand RESOLVED werden Nutzungsanfragen solange aufrechterhalten, bis die maximal benötigte Menge an Diensten verfügbar ist. Somit muss bei jeder Dienstzuteilung geprüft werden, ob nun genügend Dienste genutzt werden können. Falls dies der Fall ist, so werden die Nutzungsanfragen, welche ggf. noch an andere Dienste gestellt wurden, zurückgezogen. Hierbei handelt es sich um solche Dienste, welche zwar in der Menge *wantsUse* enthalten sind, aber nicht in *uses*. Die bereits vorgestellte Methode *newUsableService* aus Listing 4-13 realisiert das beschriebene Verhalten für den Fall, dass einem *RequiredServiceReferenceSet* die Nutzungsrechte für einen weiteren Dienst erteilt wurden.

Sobald allerdings ein verwendeter Dienst ausfällt und somit der Trigger *serviceNotAssigned* von diesem Dienst aufgerufen wird, wird versucht, hierfür Ersatz zu finden. Hierzu wird wiederum eine Nutzungsanfrage an alle verfügbaren und kompatiblen Dienste gestellt, also an alle Dienste der Menge *canUse*, welche nicht in *uses* enthalten sind. Die Methode aus Listing 4-14 realisiert dieses Verhalten.

```
1  serviceNotUsableAnymore(ProvidedService ps)  {
2   this.uses.remove(ps);
3   if(this.uses.size()==this.maxNoOfRequiredRefs-1) {
4    for(ProvidedService p : this.canUse) {
5     if(!this.uses.contains(p)) {
6      requestService(p);
7     }
8    }
9   }
10 }
```

Listing 4-14: Die Methode *serviceNotUsableAnymore*

Wie in den Abschnitten deutlich geworden ist, besteht eine enge Kopplung zwischen den Automaten einzelner Elemente einer Komponente. Bislang wurden die einzelnen Automaten vorrangig autonom betrachtet und vorgestellt. Im nun folgenden Abschnitt werden die Abhängigkeiten und Interaktionen zwischen den einzelnen Automaten näher beleuchtet, um so ein besseres Verständnis für die Funktionsweise des Konfigurationsprozesses zu vermitteln, welches sich aus den einzelnen Automaten ergibt.

4.8 Beispiel einer Komponentenspezifikation

Zur Veranschaulichung der bislang vorgestellten Konzepte zur Spezifikation einer Komponente werden diese nun anhand eines kleinen Beispiels nochmals zusammengefasst. In Abbildung 4-17 ist zu diesem Zweck eine Komponente mit Hilfe der bereits bekannten Notationselemente dargestellt. Die einzelnen Spezifikationselemente werden nun anhand dieser Komponente kurz erläutert.

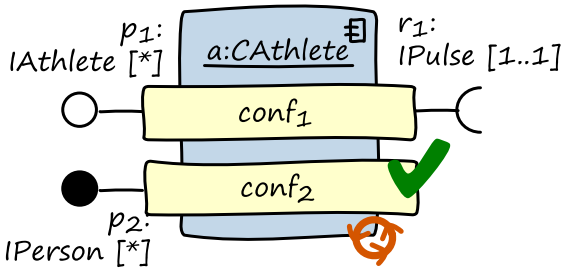


Abbildung 4-17: Beispiel zur Zusammenfassung der Spezifikationselemente

Zunächst wird für dieses Beispiel das Domänenmodell spezifiziert. Es handelt sich hierbei um ein Domänenmodell mit fünf Domänenschnittstellen:

DomainArchitecture = {BiathlonDomain}
DomainInterface = {IAthlete, IPulse, IPerson, ITrainer, IStick}
defines(BiathlonDomain) = {IAthlete, IPulse, IPerson, ITrainer, IStick}

Im nächsten Schritt werden die Elemente definiert, aus denen sich die Komponente zusammensetzt:

DynamicAdaptiveComponent = {a}
ComponentConfiguration = {$conf_1$_a, $conf_2$_a}
ProvidedService = {p_1_a, p_2_a}
RequiredServiceReferenceSet = {r_1_a}

Aus Gründen der Übersichtlichkeit wird in den Abbildungen lediglich der erste Teil des Bezeichners angegeben, da sich aus der Abbildung bereits eindeutig erschließt, zu welcher Komponente ein Element gehört.

Diese Elemente müssen nun miteinander in Beziehung gesetzt werden:

contains(a) = {$conf_1$_a, $conf_2$_a}
containedBy($conf_1$_a) = a
containedBy($conf_2$_a) = a
provides($conf_1$_a) = {p_1_a}
declares($conf_1$_a) = {r_1_a}
provides($conf_2$_a) = {p_2_a}
providedBy(p_1_a) = {$conf_1$_a}
providedBy(p_2_a) = {$conf_2$_a}
declaredBy(r_1_a) = {$conf_1$_a}
implements(p_1_a) = IAthlete
implements(p_2_a) = IPerson
refersTo(r_1_a) = IPulse

Und schließlich müssen die Attribute der einzelnen Elemente belegt werden:

$maxNoOfUsers(p_1\_a) = \infty$
$maxNoOfUsers(p_2\_a) = \infty$
$minNoOfRequiredRefs(r_1\_a) = 1$
$maxNoOfRequiredRefs(r_1\_a) = 1$
$requestRun(p_1\_a) = false$
$requestRun(p_2\_a) = true$

All diese Informationen müssen vom Domänen- bzw. Komponentenentwickler bereitgestellt werden. Das Framework verwendet diese Informationen, um zur Laufzeit das gewünschte selbstorganisierende Verhalten zu realisieren.

4.9 Konfigurationsprozess eines selbstorganisierenden Systems auf Basis selbstorganisierender Komponenten

Bislang wurden die einzelnen Elemente einer Komponente (*DynamicAdaptiveComponent*, *ComponentConfiguration*, *ProvidedService* und *RequiredServiceReferenceSet*) weitestgehend unabhängig voneinander betrachtet. In diesem Abschnitt wird nun erläutert, wie die einzelnen Elemente während des Konfigurationsprozesses zusammenwirken und so ein selbstorganisierendes Systems basierend auf selbstorganisierenden Komponenten entsteht.

Wie bereits in den einführenden Kapiteln der Arbeit erläutert wurde, entsteht ein laufendes System, welches auf selbstorganisierenden Komponenten beruht, nicht durch das Starten einer Anwendung im klassischen Sinne. Vielmehr ergibt sich ein laufendes System Bottom-Up durch das Starten einzelner, sich selbst organisierender Komponenten. Diese verbinden sich zur Laufzeit automatisch mit Diensten anderer Komponenten, so dass schließlich ein Netzwerk interagierender Komponenten und somit ein laufendes System entsteht.

Im nun folgenden Abschnitt wird zunächst erläutert, wie hierbei die Zustandsautomaten sämtlicher Elemente einer Komponente (*DynamicAdaptiveComponent*, *ComponentConfigurations*, *ProvidedServices* und *RequiredServiceReferenceSets*) zusammenwirken. Hierbei wird zunächst der Fokus auf die Erläuterung der komponenteninternen Abläufe gesetzt. Im darauffolgenden Abschnitt wird dann insbesondere die Interaktion zwischen Komponenten betrachtet und anhand eines Beispiels beschrieben.

4.9.1 Komponenteninterner Konfigurationsprozess

In diesem Abschnitt werden die komponenteninternen Abläufe während des Konfigurationsprozesses näher beleuchtet. Hierzu werden die einzelnen Konfigurationsschritte anhand einer Trainerkomponente beschrieben, welche in Abbildung 4-18 dargestellt ist.

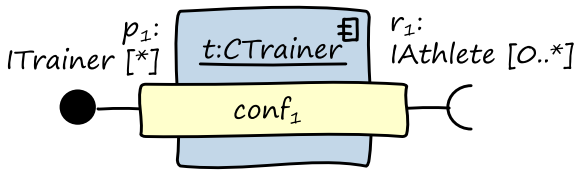


Abbildung 4-18: Beispiel einer Trainerkomponente mit einem *ProvidedService* und einem *RequiredServiceReferenceSet*

Die Komponente definiert eine *ComponentConfiguration* mit einem *ProvidedService* sowie einem *RequiredServiceReferenceSet*. Der *ProvidedService* p_1 implementiert hierbei die Schnittstelle *ITrainer* und kann von beliebig vielen Nutzern gleichzeitig verwendet werden. Zudem ist für diesen Dienst das Flag *requestRun* auf *true* gesetzt. Das *RequiredServiceReferenceSet* r_1 auf der anderen Seite definiert eine Abhängigkeit zu beliebig vielen Diensten, welche die Domänenschnittstelle *IAthlete* implementieren. Hierbei gilt die Abhängigkeit auch dann als aufgelöst, falls kein solcher Dienst zur Verfügung steht.

Der Konfigurationsprozess, welcher bei Installation dieser Komponente im System vollzogen wird, ist in Abbildung 4-19 in Form eines UML-Sequenzdiagramms dargestellt.

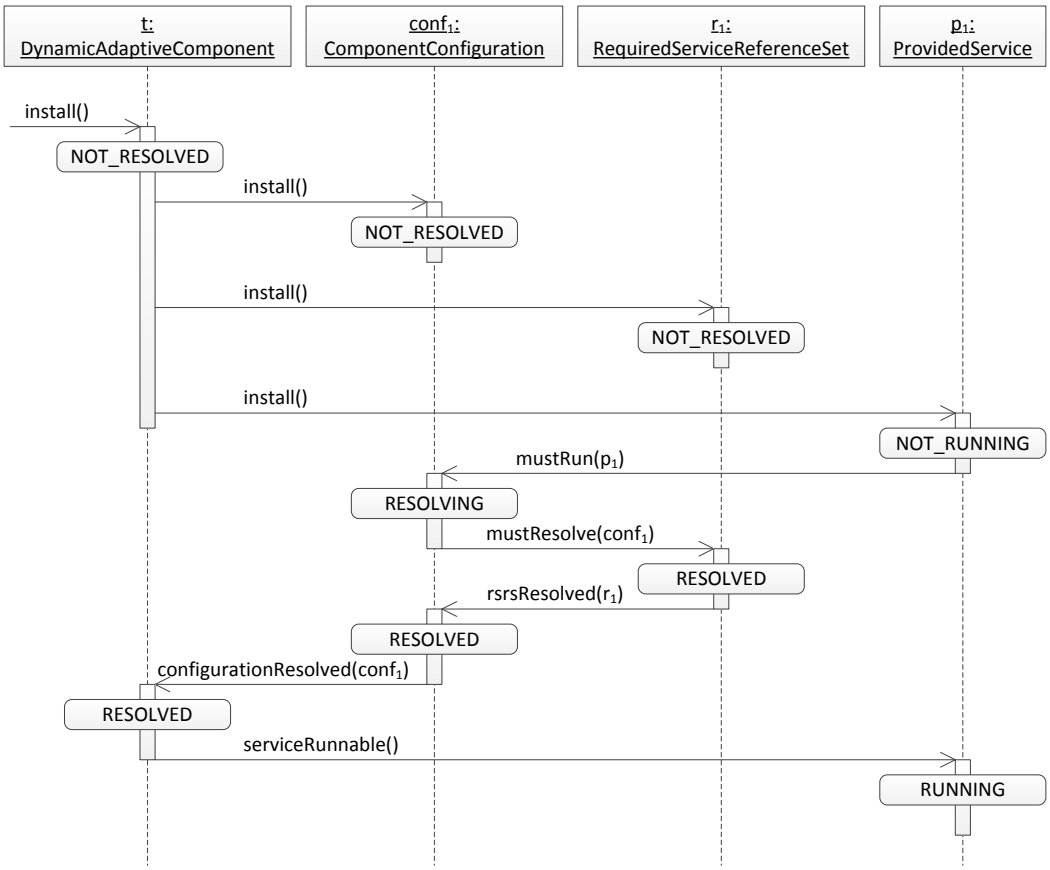


Abbildung 4-19: Komponenteninterner Konfigurationsprozess nach Installation der Komponente *t*

Der Lebenszyklus einer Komponente beginnt mit der Installation einer selbst-konfigurierenden Komponente im System. Wie bereits in den vorangegangenen Abschnitten beschrieben wurde, geschieht dies durch Aufruf der Methode *install* bei einer *DynamicAdaptiveComponent*. Daraufhin werden die Zustandsautomaten sämtlicher Komponentenelemente in ihren Ausgangszustand versetzt.

Der Automat der *DynamicAdaptiveComponent t* verbleibt solange im Zustand NOT_RESOLVED, bis von der *ComponentConfiguration* $conf_1$ der Trigger *configurationResolved* aufgerufen wird (siehe Abbildung 4-9). Die *ComponentConfiguration* $conf_1$ wiederum wartet auf den Aufruf des Triggers *mustRun*, um in einen Folgezustand wechseln zu können (siehe Abbildung 4-11). Und der Automat des *RequiredServiceReferenceSets* r_1 verbleibt solange im Zustand NOT_RESOLVED,

bis der Trigger *mustResolve* von der *ComponentConfiguration* $conf_1$ aufgerufen wird (siehe Abbildung 4-16).

An dieser Stelle kommt dem *ProvidedService* p_1 eine entscheidende Rolle zu. Beim Übergang in den Initialzustand NOT_RUNNING ruft dieser bei allen verbundenen *ComponentConfigurations* den Trigger *mustRun* auf, für den Fall, dass das Flag *requestRun* auf *true* gesetzt wurde (siehe Listing 4-4 sowie Abbildung 4-13). Dies ist für die Komponente *t* der Fall.

Wäre für den Dienst p_1 das Flag *requestRun* nicht gesetzt, so wäre der Konfigurationsprozess an dieser Stelle beendet. Der Prozess würde erst dann wieder in Gang gesetzt werden, wenn ein Verwender des Dienstes p_1 in Form einer anderen Komponente verfügbar ist. Dies verdeutlicht nochmals die Bedeutung des Flags *requestRun* der Klasse *ProvidedService*. In einem System, in dem nur Komponenten existieren, die ausschließlich Dienste ohne gesetztes Flag *requestRun* anbieten, wird niemals ein Dienst in den Zustand RUNNING überführt. Dienste werden somit nur dann gestartet, wenn hierfür die Notwendigkeit besteht. Dienste mit gesetztem *requestRun*-Flag bilden den Ausgangspunkt eines jeden Konfigurationsprozesses.

Die nun folgenden Schritte sind hingegen unabhängig davon, wieso ein Dienst ausgeführt werden möchte (ob aufgrund eines gesetzten *requestRun*-Flags, oder aufgrund der Tatsache, dass ein Nutzungswunsch einer oder mehrerer anderen Komponenten vorliegt). In beiden Fällen werden alle verbundenen *ComponentConfigurations* über den Trigger *mustRun* über den Ausführungswunsch informiert.

Der Automat der *ComponentConfiguration* $conf_1$ reagiert auf den Aufruf des Triggers *mustRun*, indem er in den Zustand RESOLVING wechselt und gleichzeitig das *RequiredServiceReferenceSet* r_1 durch Aufruf des Triggers *mustResolve* zum Auflösen der Abhängigkeiten auffordert. Würde die Komponentenkonfiguration kein *RequiredServiceReferenceSet* definieren, so würde der Automat direkt in den Zustand RESOLVED wechseln und die *DynamicAdaptiveComponent t* hierüber durch Aufruf von *configurationResolved* informieren.

Im nächsten Schritt müssen nun die Abhängigkeiten, welche durch das *RequiredServiceReferenceSet* r_1 definiert sind, aufgelöst werden. Im Beispiel werden hierzu keinerlei Dienste anderer Komponenten benötigt, da das Attribut *minNoOfRequiredRefs* auf den Wert Null gesetzt wurde. Nach Aufruf des Triggers *mustResolve* wechselt der entsprechende Automat daher direkt in den Zustand RESOLVED. Gleichzeitig wird die *ComponentConfiguration* $conf_1$ durch Aufruf der Methode *rsrsResolved* darüber informiert, dass das *RequiredServiceReferenceSet* r_1 in den Zustand RESOLVED gewechselt ist und deren Abhängigkeiten somit als aufgelöst gelten.

Die *ComponentConfiguration* $conf_1$ wechselt daraufhin in den Zustand RESOLVED, da für diese Komponentenkonfiguration alle Abhängigkeiten aufgelöst werden konnten. Gleichzeitig wird die *DynamicAdaptiveComponent t* durch Aufruf von *configurationResolved* über diesen Zustandswechsel informiert.

Diese wechselt somit in den Zustand RESOLVED und teilt denjenigen Diensten, welche durch die *ComponentConfiguration* $conf_1$ angeboten werden mit, dass sie ausgeführt werden können. Dies geschieht durch Aufruf von *serviceRunnable* (siehe Listing 4-1).

Im Beispiel führt das dazu, dass nun der *ProvidedService* p_1 ausgeführt werden kann. Dieser wechselt daraufhin in den Zustand RUNNING, da für ihn das Flag *requestRun* gesetzt ist. Dies ist gleichzeitig der letzte Schritt des Konfigurationsprozesses bei zugrundeliegender Annahme, dass lediglich die Komponente t im System verfügbar ist. Realisiert nun der Dienst beispielsweise eine grafische Benutzungsschnittstelle, so kann diese nun angezeigt und dem Nutzer die Funktionalität des Dienstes zur Verfügung gestellt werden.

4.9.2 Komponentenübergreifender Konfigurationsprozess

Nachdem nun im vorangegangenen Abschnitt die komponenteninternen Abläufe während eines Konfigurationsprozesses etwas näher beleuchtet wurden, liegt der Fokus in diesem Abschnitt in der Darstellung komponentenübergreifender Abläufe während des Konfigurationsprozesses. Hierzu wird das Ergebnis des Konfigurationsprozesses aus Abbildung 4-19 als Ausgangspunkt herangezogen. Nun sei angenommen, dass eine Pulskomponente in das System installiert wird, wie sie in Abbildung 4-20 dargestellt ist.

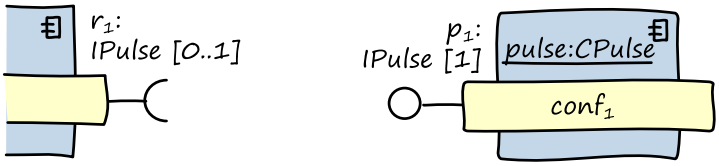


Abbildung 4-20: Beispiel einer Athletenkomponente mit einem *ProvidedService*

Die Komponente definiert eine *ComponentConfiguration* $conf_1$, welche einen Dienst p_1 anbietet, der die Domänenschnittstelle *IPulse* implementiert. Im Folgenden wird nun erläutert, wie das Framework auf die Installation dieser Komponente reagiert. Hierbei wird insbesondere die Interaktion mit einer bereits installierten Komponente beleuchtet, die den Dienst benötigt. Das UML-Sequenzdiagramm, welches in Abbildung 4-21 dargestellt ist, stellt die Abläufe grafisch dar, welche aus der Installation der Athletenkomponente resultieren.

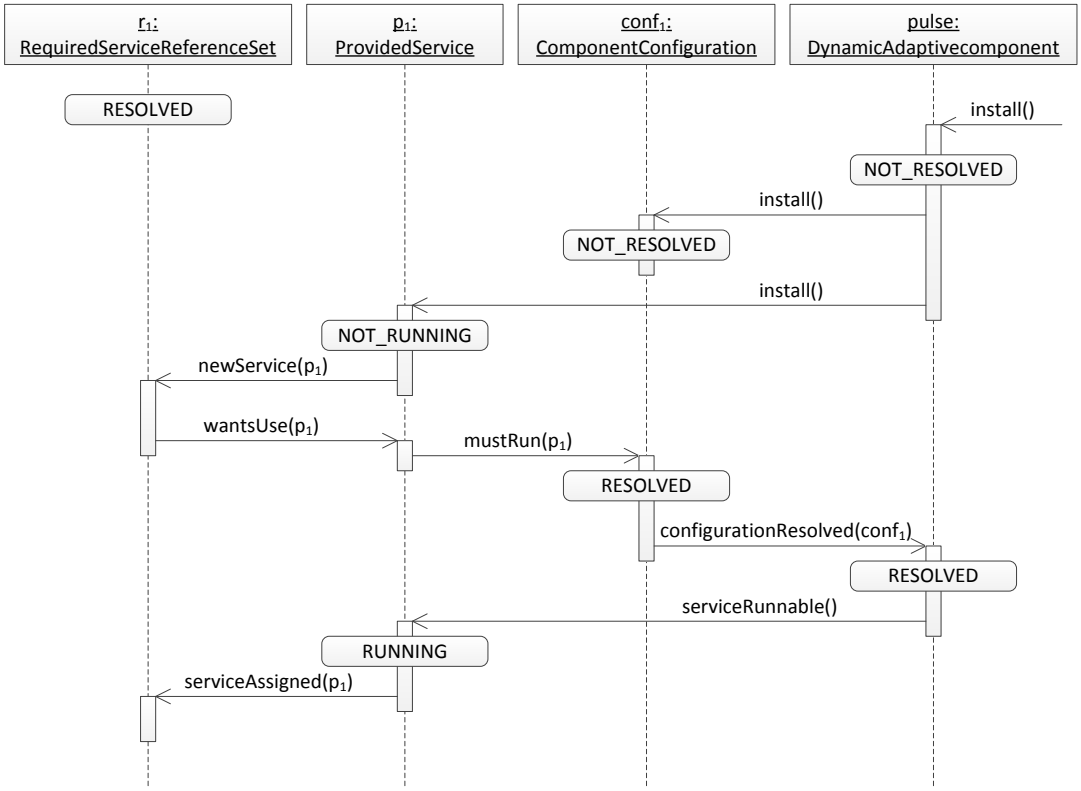


Abbildung 4-21: Komponentenübergreifender Konfigurationsprozess nach Installation der Sportlerkomponente

Es sei für das Beispiel angenommen, dass sich das *RequiredServiceReferenceSet* r_1 im Zustand RESOLVED befindet. Des Weiteren sei angenommen, dass die Komponente *pulse* im System durch Aufruf von *install* installiert wird. Die *ComponentConfiguration* $conf_1$ wird daraufhin in ihren Ausgangszustand NOT_RESOLVED versetzt. Selbiges passiert mit dem *ProvidedService* p_1. Dieser Dienst wird nun aber gleichzeitig sämtlichen im System bekannten *RequiredServiceReferenceSets* durch Aufruf von *newService* bekanntgemacht. Im konkreten Beispiel wird somit dem *RequiredServiceReferenceSet rt* mitgeteilt, dass ein neuer Dienst zur Verfügung steht.

Das Framework überprüft nun, ob dieser Dienst zur Nutzung für das *RequiredServiceReferenceSet* r_1 in Frage kommt. Dies ist der Fall, da die implementierte Domänenschnittstelle von p_1 (*IPulse*) mit derjenigen übereinstimmt, welche von r_1 referenziert wird. Entsprechend reagiert das Framework durch Aufruf von *wantsUse* beim Dienst p_1, um den Nutzungswunsch anzumelden. Von nun an muss r_1 auf die Zuteilung des Dienstes warten.

Der Dienst p_1 kann zu diesem Zeitpunkt noch nicht verwendet werden, da er sich nicht im Zustand RUNNING, sondern im Zustand NOT_RUNNING befindet. Bei Aufruf von *wantsUse* durch r_1 wird deshalb zunächst der *ComponentConfiguration* $conf_1$ durch Aufruf von *mustRun* mitgeteilt, dass der Dienst ausgeführt werden muss. Von hier ab ähnelt der Ablauf demjenigen, der bereits im vorangegangenen Abschnitt erläutert wurde. Die *ComponentConfiguration* $conf_1$ gilt unmittelbar als aktivierbar, da sie kein *RequiredServiceReferenceSet* definiert. Der *DynamicAdaptiveComponent pulse* wird durch Aufruf von *configurationResolved* mitgeteilt, dass alle Abhängigkeiten aufgelöst wurden. Diese aktiviert daraufhin die Komponentenkonfiguration und teilt dem Dienst durch Aufruf von *serviceRunnable* mit, dass er ausgeführt werden kann. Da nun für diesen Dienst eine Nutzungsanfrage vorliegt, wechselt er direkt in den Zustand

RUNNING. Gleichzeitig wird nun dem *RequiredServiceReferenceSet* das Nutzungsrecht durch Aufruf von *serviceAssigned* zugeteilt und die Komponente kann den Dienst nun nutzen.

Im nun folgenden Abschnitt wird abschließend anhand eines durchgängigen Beispiels der Konfigurationsprozess nochmals schrittweise durchgeführt, um die Abläufe innerhalb des Frameworks zu verdeutlichen.

4.10 Ablauf eines Konfigurationsprozesses am Beispiel

Zum Abschluss dieses Kapitels wird nun anhand eines durchgängigen Beispiels aus der Biathlondomäne nochmals gezeigt, wie die unterschiedlichen Elemente des Frameworks zusammenwirken. Hierbei wird zunächst mit der Installation einer Pulskomponente begonnen. Anschließend werden zwei weitere Komponenten installiert und jeweils Schrittweise die resultierenden Zustandsübergänge erläutert.

Es sei im ersten Schritt angenommen, dass lediglich eine Pulskomponente im System verfügbar ist. Diese Komponente ist in Abbildung 4-22 dargestellt.

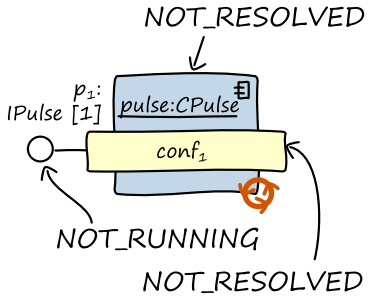


Abbildung 4-22: Die Komponente *pulse* im Initialzustand

Die Komponente besitzt eine Konfiguration mit einem angebotenen Dienst, der die Schnittstelle *IPulse* implementiert. Dieser Dienst kann von maximal einem Nutzer gleichzeitig verwendet werden. An den Elementen der Komponente ist zusätzlich der aktuelle Zustand des jeweiligen Zustandsautomaten notiert. Initial befinden sich alle Elemente in ihrem Startzustand, d.h. die Komponente befindet sich im Zustand NOT_RESOLVED, die Konfiguration im Zustand NOT_RESOLVED und der Dienst im Zustand NOT_RUNNING. Sämtliche Zustände bleiben nun solange unverändert, bis eine neue Komponente in das System kommt. Der Grund ist der, dass für den Dienst p_1 das Flag *requestRun* nicht gesetzt ist und es zudem keinen Verwender des Dienstes in Form einer anderen Komponente gibt. Deshalb besteht auch für die Konfiguration keine Notwendigkeit, Abhängigkeiten aufzulösen und die Komponente besitzt somit keine aktivierbare Konfiguration.

Im nächsten Schritt sei nun angenommen, dass eine weitere Komponente in das System kommt. Abbildung 4-23 zeigt die Situation unmittelbar nach Hinzufügen der neuen Komponente.

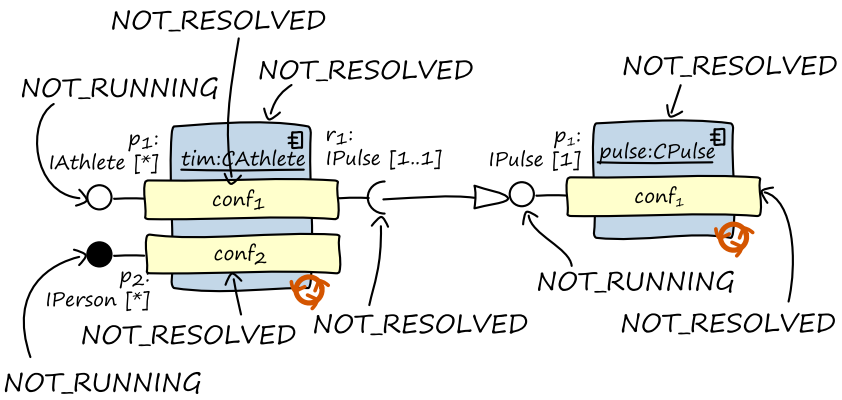


Abbildung 4-23: Eine zweite Komponente kommt hinzu

Hierbei wird zur Symbolisierung der unterschiedlichen Beziehungen zwischen *RequiredServiceReferenceSets* und *ProvidedServices* die in Abbildung 4-24 angegebene Notation verwendet.

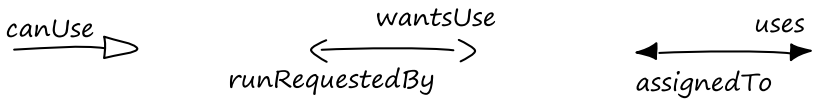


Abbildung 4-24: Legende zur Darstellung von Beziehungen zwischen Komponenten

Bei der neuen Komponente handelt es sich um eine Athletenkomponente mit zwei Konfigurationen. In der besseren Konfiguration bietet die Komponente einen Dienst an, welcher die Domänenschnittstelle *IAthlete* implementiert. Hierzu benötigt dieser Dienst Zugriff auf genau einen Dienst, welcher die Domänenschnittstelle *IPulse* implementiert. In der schlechteren Konfiguration der Komponente bietet die Komponente einen Dienst an, der die Schnittstelle *IPerson* implementiert. Diese Konfiguration besitzt keinerlei Abhängigkeiten zu Diensten anderer Komponenten. Alle Elemente der neuen Komponente befinden sich zunächst in den jeweiligen Startzuständen. Das *RequiredServiceReferenceSet* enthält ein Element in der Menge *canUse*, dargestellt durch den eingezeichneten Pfeil.

Für den Dienst p_2 der neuen Komponente ist das Flag *requestRun* gesetzt, zu erkennen am ausgefüllten Kreis. Dies hat zur Folge, dass bei Konfiguration $conf_2$ der Trigger *mustRun* aufgerufen wird (Listing 4-4). Der Dienst verbleibt nun solange im Zustand NOT_RUNNING, bis bei ihm der Trigger *serviceRunnable* aufgerufen wird. Die Konfiguration hat nun die Aufgabe, die definierten Abhängigkeiten zu Diensten anderer Komponenten aufzulösen. In diesem Fall sind keinerlei Abhängigkeiten definiert. Die Konfiguration meldet deshalb der Komponente durch Aufruf des Triggers *configurationResolved*, dass die Komponente eine aktivierbare Konfiguration besitzt. Außerdem wechselt die Konfiguration in den Zustand RESOLVED. Abbildung 4-25 zeigt diese Situation.

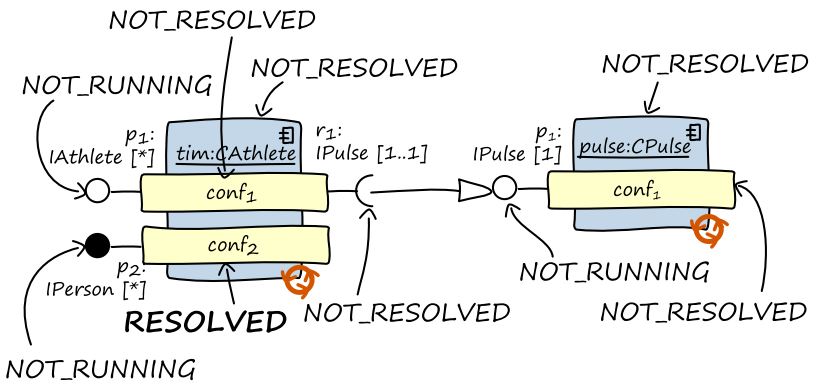


Abbildung 4-25: Die *ComponentConfiguration conf₂* wechselt in den Zustand RESOLVED

Die Komponente *tim* reagiert nun auf den Aufruf des Triggers *configurationResolved* (Listing 4-1). Zum einen fügt sie die Konfiguration der Menge *activatable* hinzu. Außerdem wird nun der Dienst p_2 durch Aufruf des Triggers *serviceRunnable* darüber informiert, dass er ausgeführt werden kann. Die Komponente wechselt anschließend in den Zustand RESOLVED. Abbildung 4-26 zeigt die neue Situation.

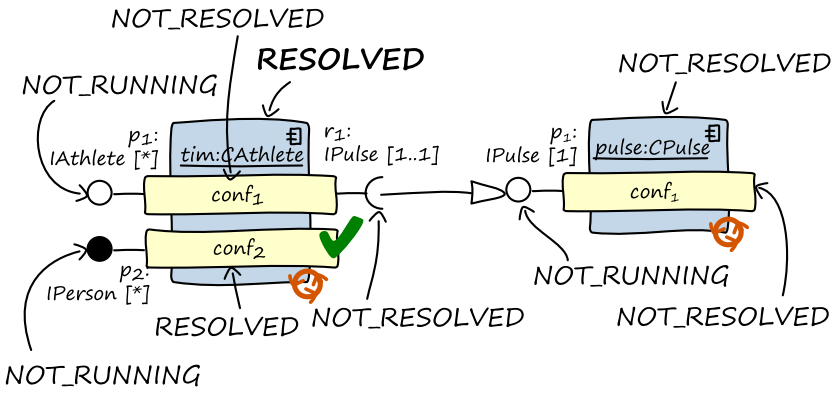


Abbildung 4-26: Die Athletenkomponente wechselt in den Zustand RESOLVED

Nun reagiert der Dienst p_2 auf den Aufruf des Triggers *serviceRunnable*. Da für den Dienst das Flag *requestRun* gesetzt ist, wechselt er vom Zustand NOT_RUNNING direkt in den Zustand RUNNING. Abbildung 4-27 zeigt das Ergebnis des Konfigurationsprozesses nach Hinzufügen der Komponente *tim*.

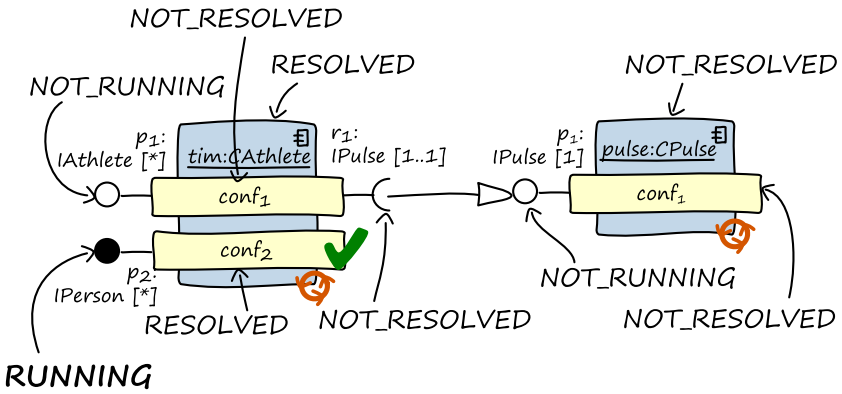


Abbildung 4-27: Das Ergebnis des Konfigurationsprozesses nach Hinzufügen der Athletenkomponente

Wie hier zu sehen ist, hat das Hinzufügen der Komponente *tim* keinerlei Auswirkungen auf die Komponente *pulse*. Der Grund ist der, dass für den Dienst p_1 der Pulskomponente keine Nutzer vorliegen. Auch die Komponente *tim* hat in der dargestellten Situation kein Interesse an der Nutzung dieses Dienstes, da die Konfiguration $conf_2$ keinen Dienst anbietet, welcher ausgeführt werden muss. Hier wird nochmals deutlich, dass der gesamte Konfigurationsprozess darauf ausgelegt ist, diejenigen Dienste in den Zustand RUNNING zu überführen, für die das Flag *requestRun* gesetzt ist. Dies ist im vorliegenden Beispiel gelungen.

Im nächsten Schritt wird nun eine dritte Komponente dem System hinzugefügt. Hierbei handelt es sich um eine Trainerkomponente mit einer Konfiguration, wie in Abbildung 4-28 dargestellt.

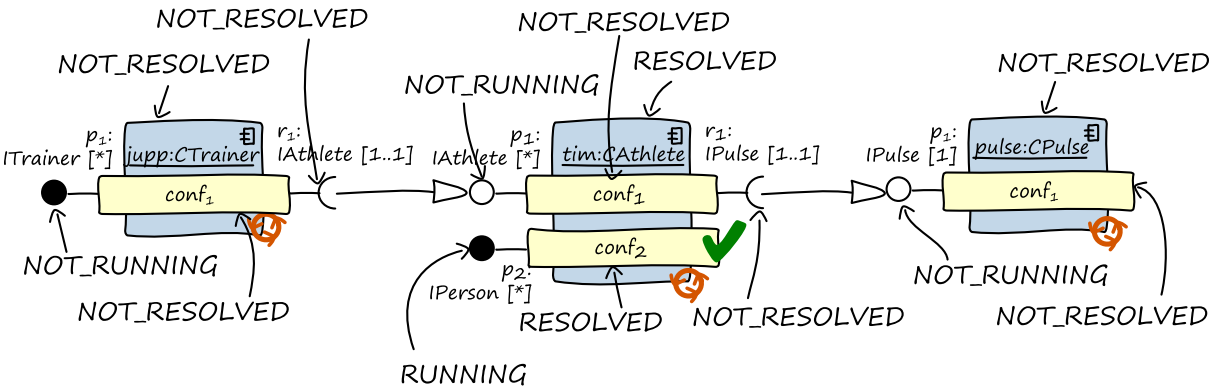


Abbildung 4-28: Eine dritte Komponente wird hinzugefügt

Zu Beginn befinden sich wieder alle Komponentenbestandteile in ihrem jeweiligen Ausgangszustand. Das *RequiredServiceReferenceSet* r_1 der neuen Komponente hat den Dienst p_1 der Athletenkomponente in die Menge *canUse* aufgenommen, da die implementierte Domänenschnittstelle mit der referenzierten Domänenschnittstelle des *RequiredServiceReferenceSets* übereinstimmt.

Die Komponente bietet einen Dienst an, für den das Flag *requestRun* gesetzt ist. Ausgehend von diesem Dienst wird nun eine Reihe von Aktionen ausgeführt, welche ihn in den Zustand RUNNING versetzen. Die Reihe der nun folgenden Konfigurationsschritte beginnt damit, dass der Dienst p_1 der Trainerkomponente auf Grund des gesetzten *requestRun*-Flags den Trigger *mustRun* bei der *ComponentConfiguration* $conf_1$ aufruft. Diese Konfiguration kann in diesem Fall nicht direkt in den Zustand RESOLVED wechseln, da sie ein mit einem *RequiredServiceReferenceSet* verbunden ist. Die Konfiguration wechselt stattdessen in den Zustand RESOLVING. Dem *RequiredServiceReferenceSet* wird hierbei durch Aufruf des Triggers *mustResolve* mitgeteilt, dass nach geeigneten Diensten gesucht werden soll. Abbildung 4-29 zeigt die entstandene Situation.

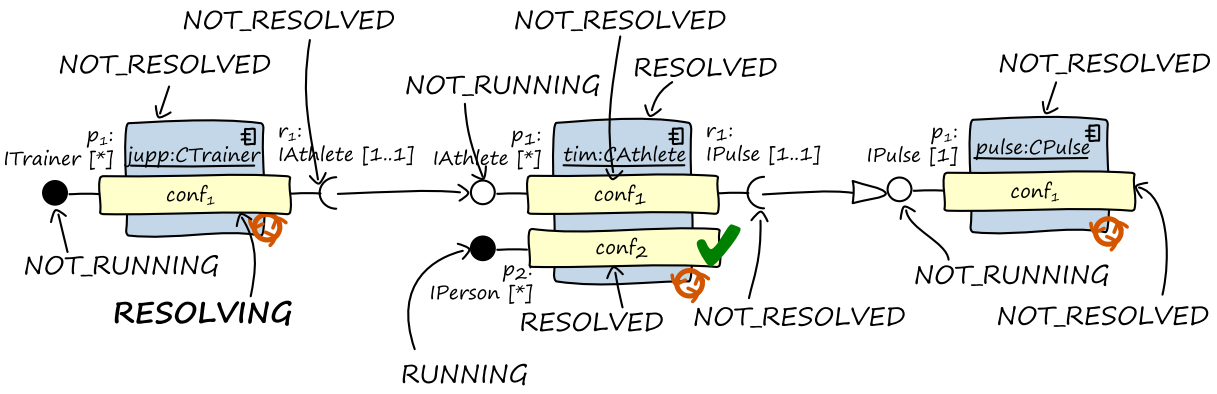


Abbildung 4-29: Die Konfiguration der Trainerkomponente wechselt in den Zustand RESOLVING

Das *RequiredServiceReferenceSet* r_1 der Komponente *jupp* signalisiert daraufhin dem Dienst p_1 der Sportlerkomponente, dass er von r_1 verwendet werden möchte. Hierfür ruft das *RequiredServiceReferenceSet* den Trigger *wantsUse* beim Dienst auf und fügt den Dienst der Menge *wantsUse* hinzu (Listing 4-11). Das *RequiredServiceReferenceSet* befindet sich anschließend im Zustand RESOLVING. Abbildung 4-30 zeigt die resultierende Situation.

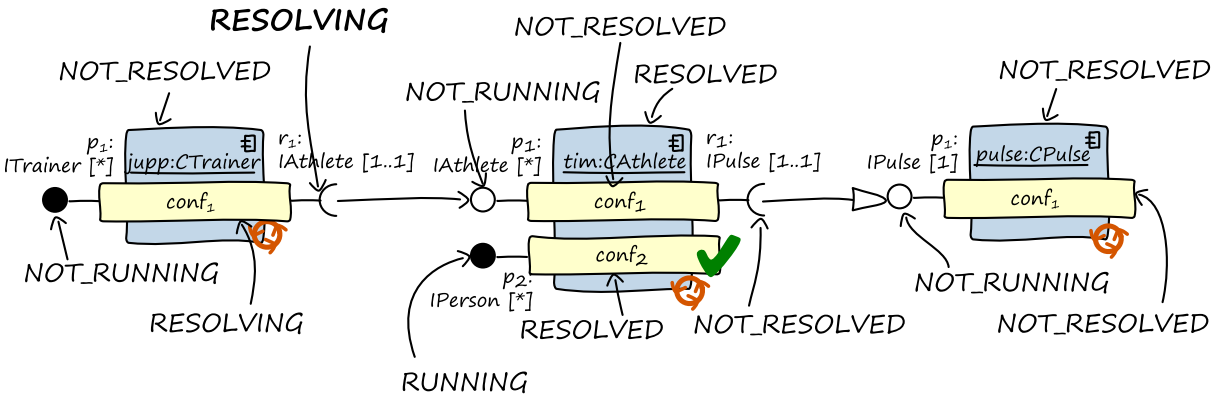


Abbildung 4-30: Das *RequiredServiceReferenceSet* r_1 der Trainerkomponente wechselt in den Zustand RESOLVING

Nun reagiert der angefragte Dienst p_1, indem er zunächst das *RequiredServiceReferenceSet* in die Menge *runRequestedBy* aufnimmt. Außerdem wird die *ComponentConfiguration* $conf_1$ der Sportler-Komponente durch Aufruf des Triggers *mustRun* darüber informiert, dass sie mit einem Dienst verbunden ist, der ausgeführt werden muss (Listing 4-5). Der Dienst verbleibt hierbei im Zustand NOT_RUNNING und wartet nun auf eine Benachrichtigung der Komponente, dass er ausgeführt werden kann.

Die *ComponentConfiguration* $conf_1$ der Athleten-Komponente wechselt bei Aufruf des Triggers *mustRun* in den Zustand RESOLVING, woraufhin das *RequiredServiceReferenceSet* r_1 durch Aufruf des Triggers *mustResolve* aufgefordert wird, die Abhängigkeiten aufzulösen. Die resultierende Situation ist in Abbildung 4-31 dargestellt.

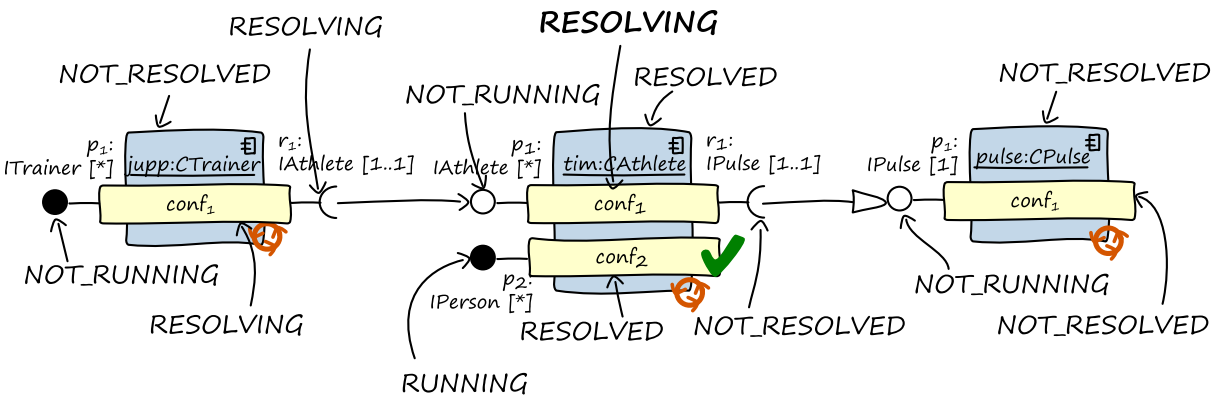


Abbildung 4-31: Die *ComponentConfiguration conf₁ der Athletenkomponente* wechselt in den Zustand RESOLVING

Nun meldet das *RequiredServiceReferenceSet* r_1 der Sportlerkomponente seinerseits Nutzungsinteresse beim Pulsdienst p_1 durch Aufruf des Triggers *wantsUse* an und fügt den Dienst der Menge *wantsUse* hinzu (Listing 4-11). Außerdem findet ein Wechsel in den Zustand RESOLVING statt. Das *RequiredServiceReferenceSet* wartet nun auf eine Zuteilung des Dienstes. Abbildung 4-32 zeigt die resultierende Situation.

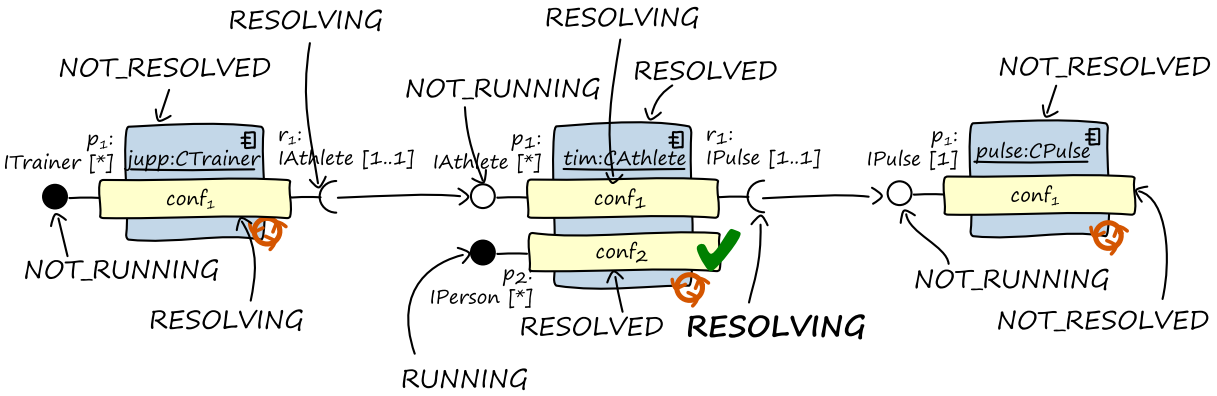


Abbildung 4-32: Das *RequiredServiceReferenceSet* der Komponente *tim* wechselt in den Zustand RESOLVING

Nun ist der Dienst p_1 der Pulskomponente am Zug, denn er muss auf den Aufruf des Triggers *wantsUse* reagieren. Hierzu nimmt er zunächst das *RequiredServiceReferenceSet* r_1 in die Menge *runRequestedBy* auf. Außerdem informiert der Dienst die Konfiguration *conf₁* darüber, dass der Dienst ausgeführt werden muss. Da die Konfiguration keinerlei Abhängigkeiten definiert, wechselt sie direkt in den Zustand RESOLVED und teilt der *DynamicAdaptiveComponent pulse* mit, dass sie nun eine aktivierbare Konfiguration besitzt. In Abbildung 4-33 ist die resultierende Situation dargestellt.

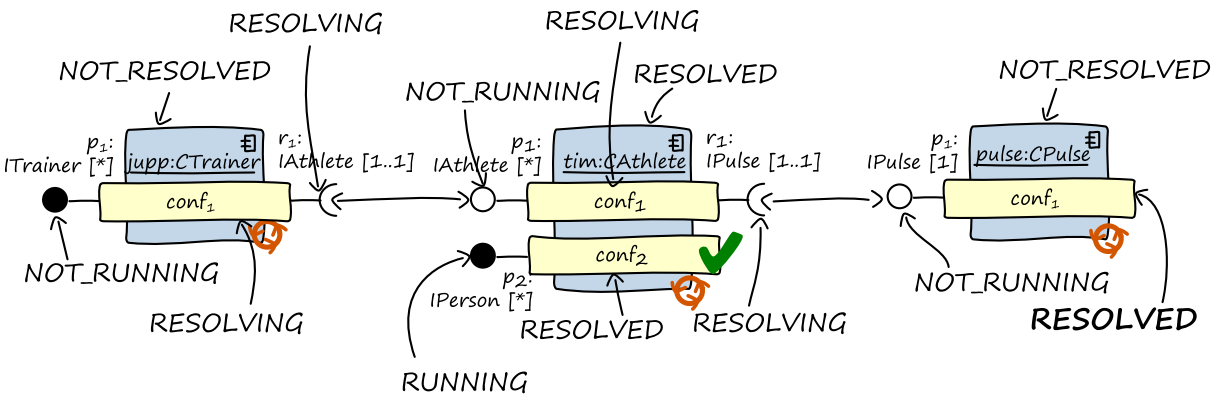


Abbildung 4-33: Die Konfiguration der Pulskomponente wechselt in den Zustand RESOLVED

Die Komponente aktiviert nun die hinzugekommene aktivierbare Konfiguration. Hierzu ruft sie beim Dienst p_1 den Trigger *serviceRunnable* auf (Listing 4-1). Außerdem wechselt die Komponente in den Zustand RESOLVED. Abbildung 4-34 zeigt die resultierende Situation.

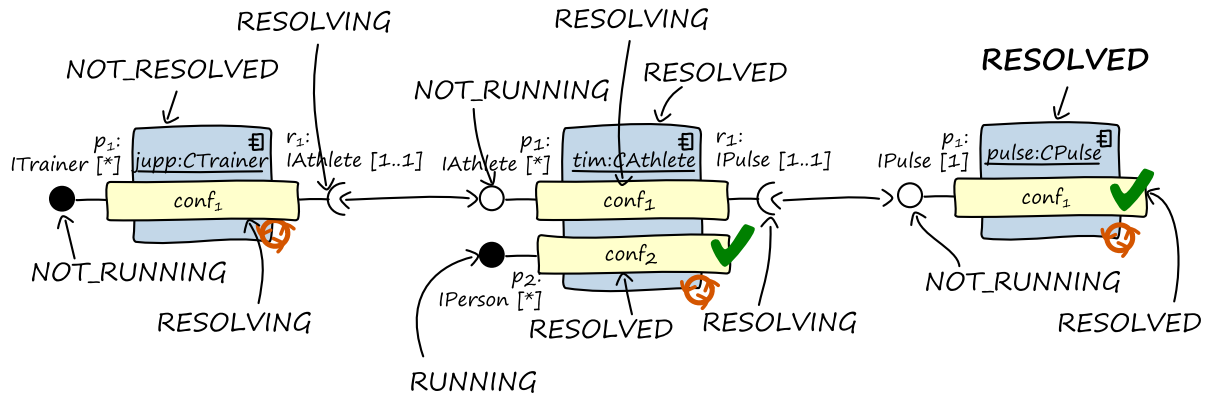


Abbildung 4-34: Die Pulskomponente wechselt in den Zustand RESOLVED

Nun beginnt die zweite Phase des Konfigurationsprozesses, indem die Dienste den *RequiredServiceReferenceSets* zur Nutzung zugeordnet werden. Es beginnt damit, dass der Pulsdienst p_1 als Reaktion auf den Trigger *serviceRunnable* in den Zustand RUNNING wechselt. Bei Eintritt in diesen Zustand wird der Dienst dem *RequiredServiceReferenceSet* r_1 zur Nutzung überlassen (Listing 4-7). Dies geschieht durch Aufruf des Triggers *serviceAssigned*. Außerdem wird das *RequiredServiceReferenceSet* der Menge *assignedTo* hinzugefügt. Das Resultat ist in Abbildung 4-35 dargestellt.

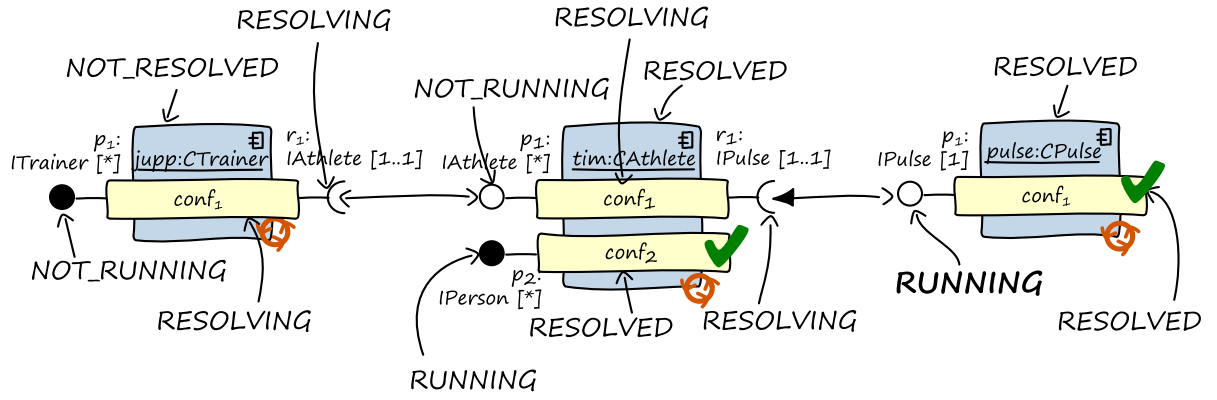


Abbildung 4-35: Der Dienst p_1 der Pulskomponente wird ausgeführt

Nun, da dem *RequiredServiceReferenceSet* ein Dienst zur Verwendung zugeteilt wurde, kann dieser in den Zustand RESOLVED wechseln, da er lediglich einen Dienst benötigt, der die Domänenschnittstelle *IPulse* implementiert. Der *ComponentConfiguration* $conf_1$ wird daraufhin durch Aufruf des Triggers *rsrsResolved* mitgeteilt, dass das *RequiredServiceReferenceSet* seine Abhängigkeiten zu Diensten anderer Komponenten aufgelöst hat. Außerdem wird der zugeteilte Dienst der Menge *uses* hinzugefügt. Abbildung 4-36 zeigt die resultierende Situation.

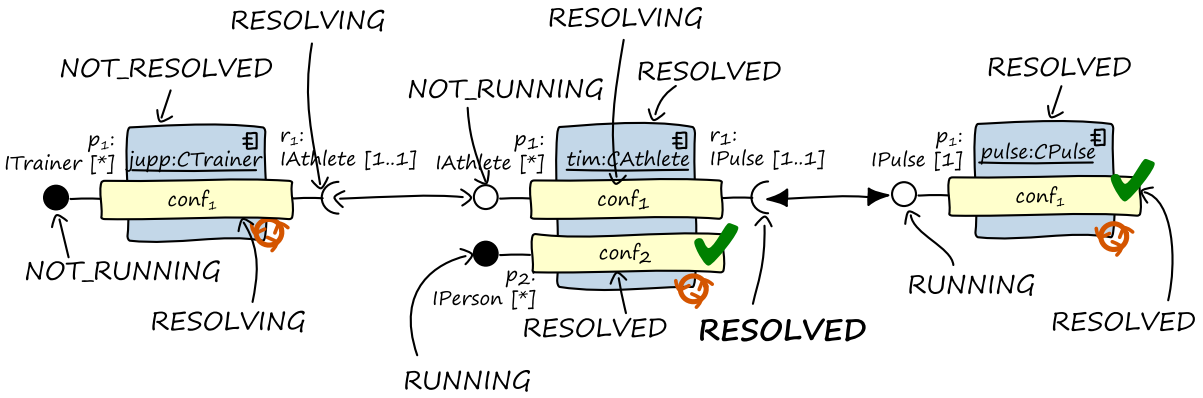


Abbildung 4-36: Das *RequiredServiceReferenceSet* der Komponente *tim* wechselt in den Zustand RESOLVED

Da die *ComponentConfiguration* $conf_1$ lediglich mit einem *RequiredServiceReferenceSet* verbunden ist, wechselt die Konfiguration in den Zustand RESOLVED und teilt gleichzeitig der Komponente über den Trigger *configurationResolved* mit, dass eine neue aktivierbare Konfiguration hinzugekommen ist. Abbildung 4-37 zeigt die resultierende Situation.

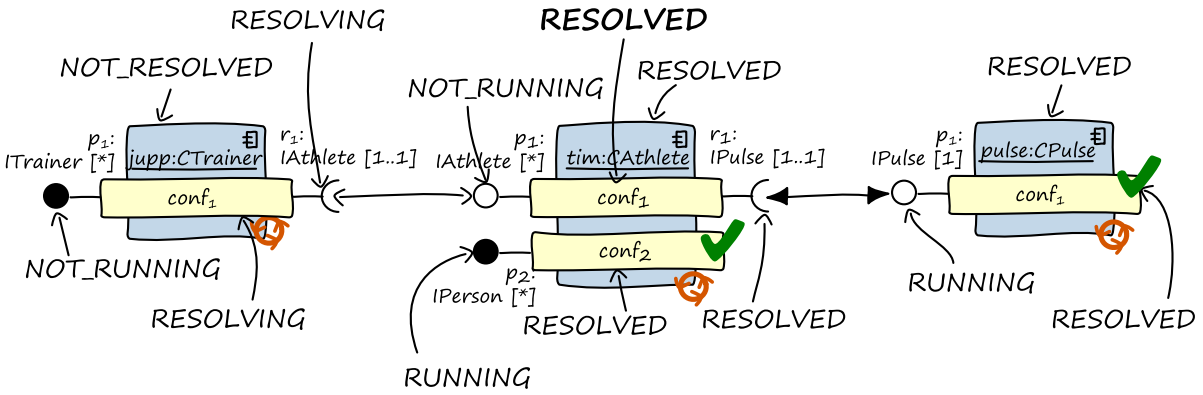


Abbildung 4-37: Die Komponente *tim* besitzt eine weitere aktivierbare Konfiguration

Nun besitzt die Komponente *tim* zwei aktivierbare Konfigurationen. Allerdings ist in der aktuellen Situation die schlechtere der beiden aktiv, nämlich $conf_2$. Aus diesem Grunde tauscht die Komponente die aktive Komponente gegen die neu aktivierbare Konfiguration aus (Listing 4-2). Hierbei wird all denjenigen Diensten, welche mit der neuen Konfiguration nicht verbunden sind, über den Trigger *serviceNotRunnable* mitgeteilt, dass sie nicht mehr ausgeführt werden können. Dies betrifft im Beispiel den Dienst p_2. Dieser Dienst wechselt daraufhin zurück in den Zustand NOT_RUNNING. Diejenigen Dienste, welche nun ausführbar geworden sind, bekommen dies durch Aufruf des Triggers *serviceRunnable* mitgeteilt. In diesem Fall ist nun der Dienst p_1 lauffähig und wechselt in den Zustand RUNNING, da für ihn eine Nutzungsanfrage vorliegt. Außerdem teilt der Dienst dem *RequiredServiceReferenceSet* r_1 der Trainerkomponente durch

Aufruf des Triggers *serviceAssigned* mit, dass er den Dienst nun verwenden darf. Das Resultat ist in Abbildung 4-38 dargestellt.

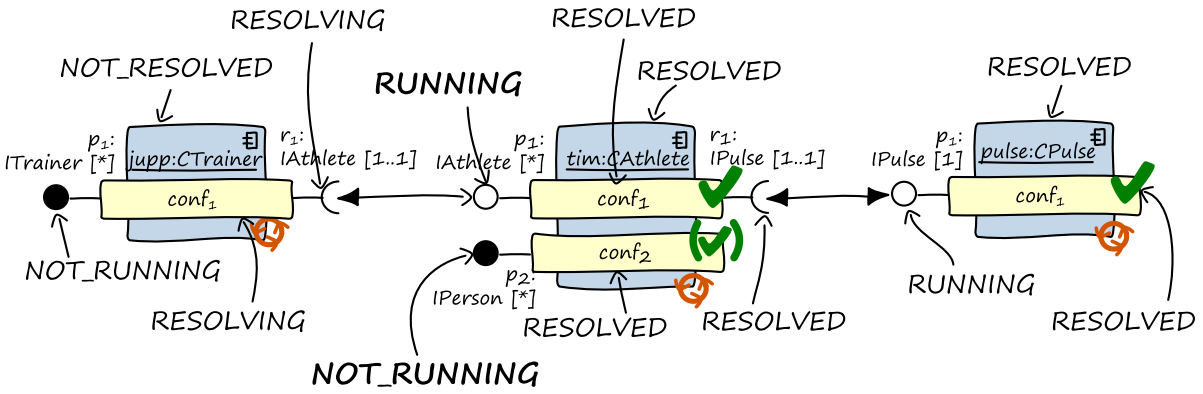


Abbildung 4-38: Austausch der aktiven Konfiguration durch eine Bessere

Nun passiert innerhalb der Komponente *jupp* das gleiche, was zuvor innerhalb der Komponente *tim* abgelaufen ist. Das *RequiredServiceReferenceSet* r_1 wechselt in den Zustand RESOLVED und teilt das der *ComponentConfiguration* $conf_1$ mit. Auch die Konfiguration kann daraufhin in den Zustand RESOLVED wechseln und teilt dies wiederum der *DynamicAdaptiveComponent jupp* mit. Sie aktiviert daraufhin die Konfiguration und ruft beim Dienst p_1 den Trigger *serviceRunnable* auf. Da für diesen Dienst das Flag *requestRun* gesetzt ist, wechselt der Dienst schließlich in den Zustand RUNNING. Abbildung 4-39 zeigt das Resultat des Konfigurationsprozesses, welcher durch Hinzufügen der Komponente *jupp* ausgelöst wurde.

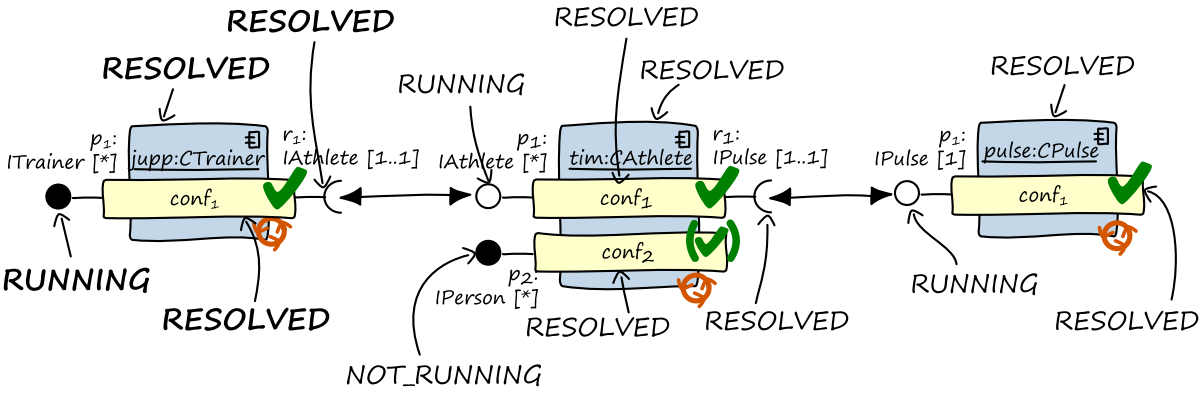


Abbildung 4-39: Resultat des Konfigurationsprozesses, welcher durch Hinzufügen der Komponente *jupp* ausgelöst wurde.

Dieses Beispiel hat sowohl gezeigt, auf welche Weise einzelne Elemente innerhalb einer Komponente miteinander interagieren um die automatische Selbstkonfiguration zu realisieren, als auch verdeutlicht, wie aus diesen ein laufendes System bestehend aus interagierenden Komponenten entsteht. In den folgenden zwei Kapiteln wird dieser Ansatz nun ausgebaut, um die Realisierung Anwendungsarchitektur-konformer Systeme basierend auf selbstkonfigurierenden Komponenten zu ermöglichen.

4.11 Zusammenfassung

In diesem Kapitel wurde ein Komponentenmodell vorgestellt, die Entwicklung und Ausführung selbstorganisierender Softwaresysteme unterstützt. Das Komponentenmodell definiert hierbei

zwei Aspekte. Zum einen wird festgelegt, aus welchen Elementen eine Komponente aufgebaut ist und wie diese zueinander in Beziehung stehen. Zum anderen definiert das Modell, wie der komponentenübergreifende Konfigurationsprozess abläuft.

Die einzelnen Elemente einer Komponente realisieren hierzu jeweils einen Zustandsautomaten, welche in diesem Kapitel ausführlich vorgestellt wurden. Die Kommunikation zwischen den Automaten wird im Wesentlichen durch den Aufruf von Triggern realisiert. Auf welche Weise diese Automaten interagieren, wurde in den vorangegangenen zwei Abschnitten anhand einiger Beispiele gezeigt.

Ein wesentlicher Aspekt ist hierbei die Kommunikation zwischen Komponenten. Hierzu kann eine Komponente beliebig viele *RequiredServiceReferenceSets* definieren. Diese spezifizieren Anforderungen an Dienste, die zur Ausführung von Diensten der eigenen Komponente benötigt werden. Diese Anforderungen beziehen sich zum einen auf die implementierte Domänenschnittstelle von Diensten, und zum anderen auf die Anzahl der Dienste.

Im nun folgenden Kapitel wird das Komponentenmodell dahingehend erweitert, dass mehr Möglichkeiten zur Spezifikation von Anforderungen an Dienste existieren. Auf diese Weise können für Komponenten detailliertere Anforderungen an Dienste anderer Komponenten definiert werden. Das im Rahmen dieser Arbeit entwickelte Framework sorgt wiederum dafür, dass diese Anforderungen automatisiert berücksichtigt werden.

5 Schnittstellenrollen

Wird ein Softwaresystem auf Basis selbstorganisierender Komponenten entwickelt, so entwickelt sich ein System, bei dem die Erfüllung von Anforderungen einzelner Komponenten im Vordergrund steht. Im vorherigen Kapitel wurde ein Framework vorgestellt, welches bereits die Berücksichtigung einiger elementarer, komponentenlokaler Anforderungen umsetzt. Einer der wesentlichen Aspekte hierbei war die Möglichkeit, Anforderungen an benötigte Dienste definieren zu können. Nur beschränken sich die Möglichkeiten zunächst darauf, die Domänenschnittstelle zu bestimmen, die von anderen Anbietern implementiert werden muss, sowie festzulegen, wie viele Referenzen auf unterschiedliche Anbieter für eine Komponente benötigt wurden. So konnte für eine Athleten-Komponente beispielsweise spezifiziert werden, dass sie genau eine Referenz auf einen Dienst benötigt, der die Domänenschnittstelle *IPulse* implementiert. In vielen Fällen reichen diese elementaren Möglichkeiten jedoch nicht aus. Deshalb werden die Möglichkeiten hinsichtlich der Spezifikation von Kriterien an zu verwendende Dienste in diesem Kapitel erweitert. Hierzu wird im folgenden Abschnitt erläutert, um welche Möglichkeiten es sich handelt. In den darauffolgenden Abschnitten wird dann beschrieben, wie das Framework aus Kapitel 4 hierzu erweitert werden muss.

5.1 Motivation

Abhängigkeiten einer Komponente zu Diensten anderer Komponenten werden im vorliegenden Framework durch die Klasse *RequiredServiceReferenceSet* repräsentiert. Jedes *RequiredServiceReferenceSet* definierte hierbei Abhängigkeiten zu Diensten, welche eine bestimmte Domänenschnittstelle implementieren. Über die Relation *refersTo* wurde für ein *RequiredServiceReferenceSet* festgelegt, um welche Domänenschnittstelle es sich hierbei handelt. Die Anforderungen, welche ein *RequiredServiceReferenceSet* definiert, bezogen sich des Weiteren auf die minimale und maximale Anzahl benötigter Referenzen auf Dienste anderer Komponenten. Sobald die minimale Anzahl an Diensten verfügbar war, galten die Abhängigkeiten des *RequiredServiceReferenceSet* als aufgelöst.

Diese Möglichkeiten reichen jedoch in vielen Fällen nicht aus, eine Komponente durch das Framework mit geeigneten Diensten zu versorgen. Zur Veranschaulichung sei hierzu eine Sportlerkomponente mit einer *ComponentConfiguration* und einem *ProvidedService* angenommen, wobei für den Dienst das Flag *requestRun* gesetzt ist. Zusätzlich definiert die Komponente zwei *RequiredServiceReferenceSets*. Die Komponente ist in Abbildung 5-1 dargestellt.

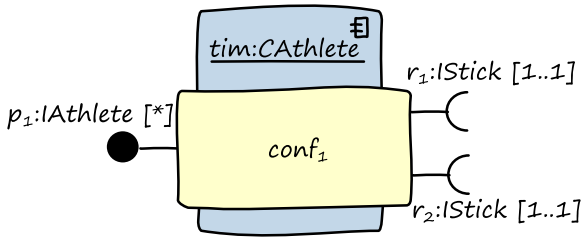


Abbildung 5-1: Eine Sportlerkomponente mit einem *ProvidedService* und zwei *RequiredServiceReferenceSets*

Der angebotene Dienst p_1 der Komponente soll Informationen zur Lauftechnik des Sportlers liefern. So soll der Dienst beispielsweise analysieren, ob der Sportler den Doppelstockschub verwendet, oder im Diagonalschritt läuft und das Ergebnis der Analyse über die Schnittstelle *IAthlete* beispielsweise Trainerkomponenten bereitstellen.

Damit der *ProvidedService* p_1 diese Funktionalität erbringen kann, muss er Zugriff auf Daten besitzen, welche Rückschlüsse auf die Stockbewegungen zulassen. In diesem Fall sei angenommen, dass über die Schnittstelle *IStick* Beschleunigungswerte für die x-, y- und z-Achse eines Skistocks abgefragt werden können. Zudem benötigt der Dienst Zugriff auf die Daten sowohl des linken als auch des rechten Skistocks. Diese Abhängigkeiten wurden für die Komponente *tim* aus Abbildung 5-1 mit Hilfe zweier *RequiredServiceReferenceSets* spezifiziert.

Würde sich diese Komponente unter Verwendung des im vorherigen Kapitel vorgestellten Frameworks konfigurieren, so würde der Dienst p_1 genau dann gestartet werden, wenn sowohl das *RequiredServiceReferenceSet* r_1 als auch r_2 mit einem Dienst verbunden sind, der die Schnittstelle *IStick* implementiert. Und an dieser Stelle wird verdeutlicht, weshalb die bisherigen Möglichkeiten zur Spezifikation von Anforderungen an benötigte Dienste nicht immer ausreichen. So würde nämlich der Dienst p_1 auch dann gestartet werden, wenn sowohl r_1 als auch r_2 mit einem linken Skistock verbunden wären. Und dies würde zu einer falschen Analyse der Lauftechnik führen.

Die Domänenschnittstelle *IStick* definiert zum Abfragen der Seite, auf der ein Skistock eingesetzt wird, die Methode *getSide():String*. Ein Dienst, der diese Schnittstelle implementiert, liefert *left* zurück, falls er die Daten eines linken Skistocks bereitstellt, und *right*, falls er Bewegungsdaten für einen rechten Skistock liefert. Derlei Informationen werden allerdings bislang bei der Entscheidung, welcher Dienst mit welchem *RequiredServiceReferenceSet* verbunden werden soll, nicht mit einbezogen. So kann innerhalb eines *RequiredServiceReferenceSets* nicht definiert werden, dass die Abhängigkeit nur dann als aufgelöst angenommen werden kann, wenn der verwendete Dienst bei Aufruf von *getSide* den Wert *left* bzw. *right* zurückliefert.

Es lassen sich zahlreiche weitere Beispiele finden, in denen die Berücksichtigung von Rückgabewerten bei Methoden von Domänenschnittstellen unerlässlich ist, um eine korrekte Realisierung angebotener Funktionalität zu gewährleisten. Aus diesem Grund wird das bereits vorgestellte Komponentenmodell in diesem Kapitel dahingehend erweitert. Hierzu wird ein neues Konzept, das der Schnittstellenrollen, eingeführt, und in das Komponentenmodell integriert. Alle übrigen Eigenschaften des bestehenden Modells bleiben dabei erhalten. D.h. das System konfiguriert sich weiterhin automatisch, basierend auf selbstorganisierenden Komponenten, allerdings nun auch unter Berücksichtigung von Rückgabewerten von verwendeten Diensten.

5.2 Erweiterung des Komponentenmodells

Um bei der Auflösung von Abhängigkeiten auch Rückgabewerte von Domänenschnittstellen-Methoden zu berücksichtigen, muss das bereits vorgestellte Komponentenmodell aus Kapitel 4 erweitert werden. Diese Erweiterungen werden nun im Folgenden beschrieben. Hierbei muss zum einen die Möglichkeit gegeben werden, derartige Abhängigkeiten spezifizieren zu können. Zum anderen muss das Framework dahingehend angepasst werden, dass nun auch diese neu

hinzugekommenen Anforderungen automatisch während des Konfigurationsprozesses berücksichtigt werden.

Im hier vorliegenden Lösungsansatz wird zu diesem Zweck die Domänenarchitektur erweitert. Bislang bestand diese ausschließlich aus Domänenschnittstellen inklusive der darin spezifizierten Syntax und Semantik sowohl der Schnittstellen als auch der darin definierten Methoden. Diese Domänenschnittstellen definieren im Grunde einen Vertrag zwischen Dienstnutzer und Dienstanbieter. Ein Dienstanbieter verpflichtet sich hierbei, eine Domänenschnittstelle gemäß der definierten Syntax und Semantik zu implementieren. Dienstnutzer auf der anderen Seite können entsprechend annehmen, dass die Schnittstelle konform zur definierten Syntax und Semantik implementiert ist. Der Lösungsansatz basiert nun darauf, diese Vertragsspezifikation zu erweitern. Hierzu wird ein Konzept eingeführt, welches im Folgenden mit *Schnittstellenrollen* bezeichnet wird. Eine Schnittstellenrolle referenziert genau eine Domänenschnittstelle und kann zusätzlich Bedingungen an Rückgabewerte von Methoden enthalten. Ein angebotener Dienst realisiert eine Schnittstellenrolle nur dann, wenn er sowohl die referenzierte Domänenschnittstelle implementiert, als auch die zusätzlich definierten Bedingungen erfüllt.

Wie bereits erwähnt beziehen sich diese Bedingungen auf die Rückgabewerte von Methoden in der referenzierten Domänenschnittstelle. Diese können sich im Laufe der Ausführung eines Dienstes ändern. So kann eine Skistockkomponente, welche die Domänenschnittstelle *IStick* implementiert, zu einem Zeitpunkt den Wert *left* bei Aufruf von *getSide* zurückgeben, und zu einem späteren Zeitpunkt *right*. Die Konformität eines Dienstes lässt sich somit zum einen erst zur Laufzeit ermitteln, und zum anderen muss die Konformität stetig neu überprüft werden.

Bevor das Konzept im Detail vorgestellt wird, wird dessen Funktionsweise nun kurz anhand der Komponente, welche bereits aus Abbildung 5-1 bekannt ist, erläutert. Die Komponente benötigt eine Referenz auf einen linken, und eine auf einen rechten Skistock. Hierzu wird das Domänenmodell um die Schnittstellenrollen *LeftStickRole* und *RightStickRole* erweitert. Beide verweisen auf die Domänenschnittstelle *IStick*. Innerhalb der Schnittstellenrolle *LeftStickRole* wird zusätzlich hinterlegt, dass ein Dienst nur dann konform zur Schnittstellenrolle ist, wenn er bei Aufruf von *getSide* den Wert *left* zurückliefert. Für die Rolle *RightStickRole* wird ebenfalls eine entsprechende Bedingung spezifiziert. Nun kann das *RequiredServiceReferenceSet* r_1 der Komponente *tim* aus Abbildung 5-1 beispielsweise auf die Rolle *LeftStickRole* verweisen, und r_2 auf *RightStickRole*. Im vorliegenden Lösungsansatz sorgt das Framework nun dafür, dass die *RequiredServiceReferenceSets* r_1 und r_2 nur mit jeweils einem Skistockdienst verbunden werden, der konform zur jeweils referenzierten Schnittstellenrolle ist.

Zur Realisierung dieses Lösungsansatzes wird das bereits aus Kapitel 4 bekannte Komponentenmodell erweitert um die Klasse *InterfaceRole*. Diese repräsentiert das Konzept der Schnittstellenrolle im Modell. Die Erweiterung des Komponentenmodells inklusive der hinzugekommenen Assoziationen ist in Abbildung 5-2 dargestellt und wird nun im weiteren Verlauf ausführlich vorgestellt.

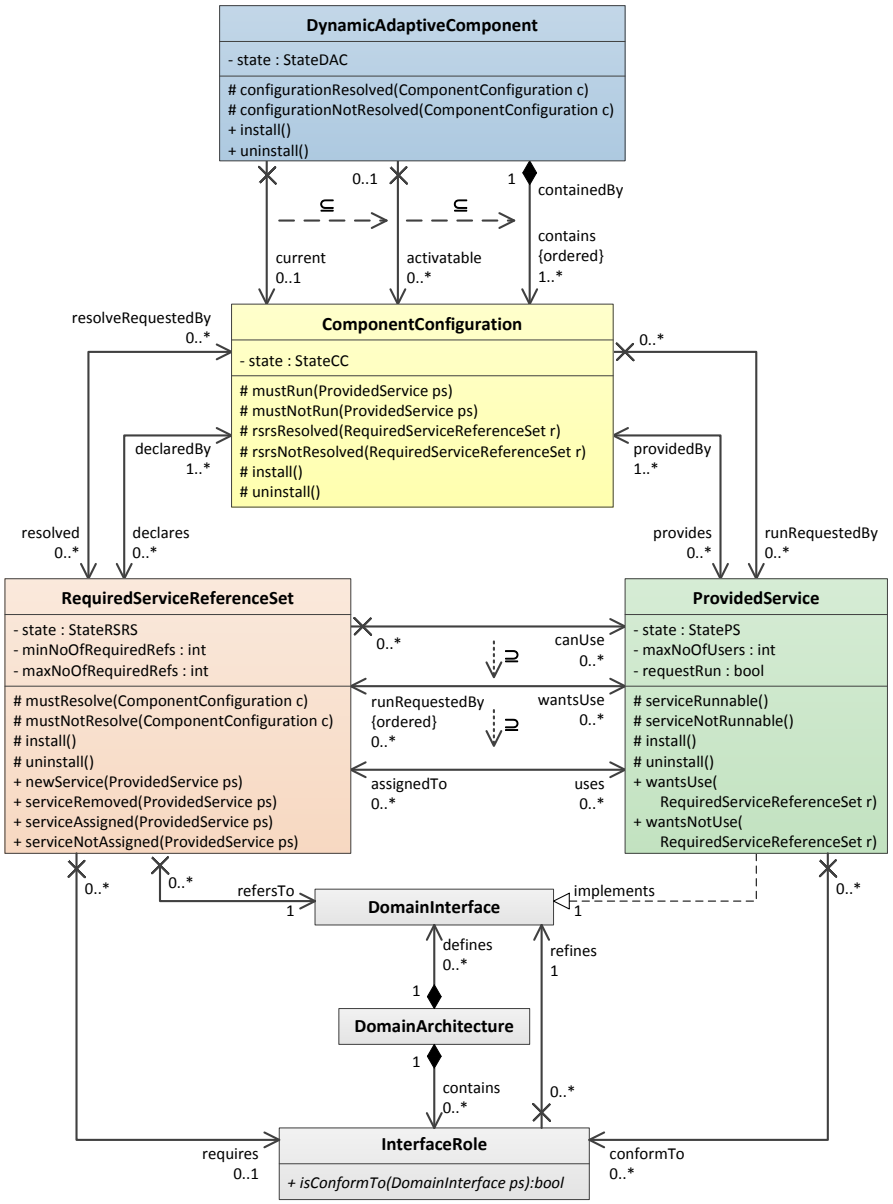


Abbildung 5-2: Erweiterung des Komponentenmodells um Schnittstellenrollen

Schnittstellenrollen sind, wie bereits erwähnt, Teil der Domänenarchitektur. Sie gehören somit zu dem Teil des Modells, der sowohl Dienstnutzern als auch Dienstanbietern bereits zur Entwicklungszeit bekannt ist. Innerhalb einer Domänenarchitektur können beliebig viele Schnittstellenrollen definiert werden. Jede Instanz der Klasse *InterfaceRole* repräsentiert genau eine Schnittstellenrolle. Diejenigen Rollen, welche innerhalb einer Domänenarchitektur enthalten sind, sind im Modell über die Assoziation *contains* repräsentiert (siehe Formel 5-1):

$$\boldsymbol{contains}: DomainArchitecture \rightarrow \mathcal{P}(InterfaceRole)$$

Formel 5-1

Wie bereits eingangs erläutert wurde, können mit Hilfe einer Schnittstellenrolle Bedingungen formuliert werden, die über diejenigen hinausgehen, welche bereits innerhalb einer Domänenschnittstelle spezifiziert werden. Jede *InterfaceRole* referenziert hierbei genau ein *DomainInterface* und verfeinert die dort hinterlegten Bedingungen. Diese referenzierte

Domänenschnittstelle ist im Modell durch die Assoziation *refines* repräsentiert (siehe Formel 5-2):

$$\boldsymbol{refines}: InterfaceRole \rightarrow DomainInterface$$

Formel 5-2

Hierbei können sich durchaus mehrere *InterfaceRoles* auf eine Domänenschnittstelle beziehen, so wie dies z.B. Fall der Athletenkomponente *tim* benötigt wird.

RequiredServiceReferenceSets können diese Schnittstellenrollen nun nutzen, um so Anforderungen an Dienste zu stellen, die über das Referenzieren von Domänenschnittstellen hinausgehen. So kann ein *RequiredServiceReferenceSet* beispielsweise zusätzlich zur Schnittstelle *IStick* eine Schnittstellenrolle *LeftStickRole* referenzieren. Das Framework stellt zur Laufzeit sicher, dass die Komponente über dieses *RequiredServiceReferenceSet* ausschließlich mit Diensten verbunden wird, welche die Schnittstelle *IStick* implementieren, und gleichzeitig einen linken Skistock repräsentieren. Auf die Spezifikation letzterer Anforderung wird im Verlaufe dieses Kapitels noch ausführlich eingegangen.

Diejenige Schnittstellenrolle, welche von einem *RequiredServiceReferenceSet* referenziert wird, ist im Modell durch die Assoziation *requires* repräsentiert (siehe Formel 5-3):

$$\boldsymbol{requires}: RequiredServiceReferenceSet \rightarrow InterfaceRole$$

Formel 5-3

Hierbei ist wichtig, dass diejenige Domänenschnittstelle, welche von einem *RequiredServiceReferenceSet* referenziert wird, mit derjenigen übereinstimmt, welche von der ggf. referenzierten Schnittstellenrolle verfeinert wird (siehe Formel 5-4):

$$\forall rsrs \in RequiredServiceReferenceSet:$$
$$requires(rsrs) \neq \emptyset \Longrightarrow refersTo(rsrs) = refines\big(requires(rsrs)\big)$$

Formel 5-4

Wie bereits erwähnt, ist das Framework dafür verantwortlich, einer Komponente nur diejenigen benötigten Dienste zuzuweisen, welche konform zur ggf. referenzierten Domänenschnittstelle sind. Die Menge der konformen Dienste ist im Modell durch die Assoziation *conformTo* repräsentiert (siehe Formel 5-5):

$$\boldsymbol{conformTo}: ProvidedService \rightarrow \mathcal{P}(InterfaceRole)$$

Formel 5-5

Hierbei gilt, dass ein Dienst nur dann konform zu einer Schnittstellenrolle ist, falls die implementierte Domänenschnittstelle des Dienstes mit derjenigen übereinstimmt, welche durch die Schnittstellenrolle referenziert wird (siehe Formel 5-6):

$$\forall ps \in ProvidedService, \forall ir \in InterfaceRole:$$
$$ir \in conformTo(ps) \Longrightarrow implements(ps) = refines\big(conformTo(ps)\big)$$

Formel 5-6

Nun bleibt noch zu klären, wie innerhalb einer Schnittstellenrolle Anforderungen an den Rückgabewert von Domänenschnittstellen-Methoden spezifiziert werden können. Außerdem

muss der Konfigurationsprozess derart angepasst werden, dass diese Anforderungen berücksichtigt werden. In den folgenden zwei Abschnitten werden diese zwei Aspekte im Detail erläutert.

5.3 Spezifikation von Schnittstellenrollen

Schnittstellenrollen werden im Komponentenmodell durch die Klasse *InterfaceRole* repräsentiert. Über diese Klasse werden im Domänenmodell Anforderungen an Dienste hinterlegt, welche sich auf den Rückgabewert von Domänenschnittstellen-Methoden beziehen. Zur Spezifikation derartiger Anforderungen definiert die Klasse *InterfaceRole* eine abstrakte Methode namens *isConformTo*. Hierbei handelt es sich um ein Prädikat, welches *true* zurückliefert, falls ein Dienst alle Anforderungen erfüllt, und ansonsten *false*. Innerhalb dieser Methode werden auf diese Weise Anforderungen definiert.

Im Folgenden wird nun erläutert, welche Möglichkeiten der Anforderungsspezifikation durch diese Methode geboten werden. Zur Spezifikation der Methode wird im weiteren Verlauf des Kapitels die Sprache OCL (Object Constraint Language) verwendet [Obj06]. Hierbei handelt es sich um eine Sprache zur Spezifikation von Bedingungen, welche im Rahmen der UML eingeführt wurde. Im Rahmen der Spezifikation von Schnittstellenrollen wird diese Sprache zur Definition des Prädikats *isConformTo* verwendet. Die Signatur der Methode ist in Formel 5-7 nochmals angegeben:

$$public\ abstract\ bool\ isConformTo(DomainInterface\ di)$$

Formel 5-7

Diese Methode wird durch das Framework immer dann aufgerufen, wenn es eine Überprüfung der Konformität eines Dienstes zur entsprechenden Schnittstellenrolle für notwendig erachtet. Liefert die Methode *true* zurück, so ist die Konformität gegeben, ansonsten nicht. Der auf Konformität zu überprüfende Dienst wird hierbei der Methode als Parameter übergeben.

Damit die Konformität eines übergebenen Dienstes innerhalb der Methode überprüft werden kann, muss hierin Zugriff auf Methoden des Dienstes bestehen. Hierbei gewährt das Framework Zugriff auf diejenigen Methoden, welche in der Domänenschnittstelle definiert sind, die von der Schnittstellenrolle durch *refines* (siehe Formel 5-2) referenziert wird. So kann innerhalb der Methode *isConformTo* einer Schnittstellenrolle *LeftStickRole*, die die Domänenschnittstelle *IStick* referenziert, auf die Methode *getSide* zugegriffen werden.

Die Methoden eines Dienstes werden nicht nur vom Framework zur Überprüfung der Konformität zwischen Rolle und Dienst aufgerufen. Sie werden parallel dazu ggf. ebenfalls von anderen Komponenten aufgerufen zur Realisierung gewisser Funktionalitäten. Dies führt immer dann zu Problemen, wenn bei einem Methodenaufruf der interne Zustand des Dienstes verändert wird. Zur Verdeutlichung des Problems sei ein Dienst angenommen, welcher eine Methode *getNextValue():int* anbietet, die bei jedem Aufruf die nächstgrößere ganze Zahl gegenüber dem letzten Aufruf zurückliefert. So liefert die Methode beispielsweise beim allerersten Aufruf den Wert 0 zurück, beim nächsten den Wert 1 usw. Ein Nutzer dieses Dienstes erwartet somit, dass die Werte 0, 1 usw. nacheinander zurückgeliefert werden. Wird nun diese Methode durch das Framework zur Überprüfung der Rollenkonformität aufgerufen, so kann es vorkommen, dass der Dienstnutzer beispielsweise die Werte 0, 2, 5 usw. zurückgeliefert bekommt. Dieses Verhalten ist aus Sicht des Dienstnutzers nicht nachvollziehbar. Aus diesem

Grunde darf innerhalb der Methode *isConformTo* ausschließlich auf sogenannte seiteneffektfreie Methoden zugegriffen werden. Dies sind eben jene Methoden, welche den internen Zustand des Dienstes sowie der gesamten Komponente nicht verändern. Hierzu muss im Domänenmodell bei den Domänenschnittstellen eine Information darüber hinterlegt sein, welche Methoden seiteneffektfrei sind, und welche nicht. Diese Information wird im vorliegenden Lösungsansatz informell angegeben.

Innerhalb des Prädikats können ferner alle aus herkömmlichen Programmiersprachen bekannten Funktionen verwendet werden. Hierzu zählen z.B. sämtliche Rechenoperationen und Operationen auf Zeichenketten.

Die Definition des Prädikats *isConformTo* mit Hilfe von OCL beginnt immer mit der Angabe eines sogenannten Kontextes, in diesem Fall den Namen einer Schnittstellenrolle. Durch zwei Doppelpunkte getrennt wird anschließend die Methodensignatur angegeben. Der Rückgabewert wird innerhalb des Methodenrumpfes durch *post:result=...* angegeben. Die Definition eines Prädikates wird nun anhand eines kleinen Beispiels erläutert.

Für das Beispiel werden zwei Schnittstellenrollen definiert und in der Domänenarchitektur hinterlegt:

```
contains(BiathlonDomain) = {LeftStickRole, RightStickRole}
refines(LeftStickRole) = IStick
refines(RightStickRole) = IStick
```

Innerhalb des jeweils zu definierenden Prädikats *isConformTo* der beiden Schnittstellenrollen besteht somit Zugriff auf sämtliche Methoden der Domänenschnittstelle *IStick*, wobei angenommen wird, dass diese Schnittstelle ausschließlich seiteneffektfreie Methoden definiert.

Nun muss für die Schnittstellenrolle *LeftStickRole* spezifiziert werden, dass nur diejenigen *ProvidedServices* zu ihr konform sind, welche bei Aufruf von *getSide* die Zeichenkette *„left“* zurückgeben. Dies geschieht durch folgende, in OCL spezifizierte Methode, welche in Listing 5-1 angegeben ist:

```
1  context LeftStickRole :: isConformTo(IStick ps) : bool
2  post: result = ps.getSide()='left'
```

Listing 5-1: Spezifikation des Prädikats *isConformTo* für die Schnittstellenrolle *LeftStickRole*

Der Rückgabewert wird als Nachbedingung angegeben, welche durch *post:result=* eingeleitet wird. Dahinter wird im konkreten Fall dann *true* zurückgegeben, falls der Dienst bei Aufruf von *getSide* die Zeichenkette *left* zurückliefert. Die Einzelheiten der Sprache OCL werden in [Obj06] näher vorgestellt.

Mit den nun bekannten Konzepten lassen sich die Kriterien, unter denen ein Dienst mit einem *RequiredServiceReferenceSet* verbunden werden darf, wie in Formel 5-8 angegeben formalisieren:

$$\forall ps \in ProvidedService, \forall rsrs \in RequiredServiceReferenceSet:$$
$$ps \in uses(rsrs) \wedge requires(rsrs) \neq \emptyset \Longrightarrow requires(rsrs).isConformTo(ps)$$

Formel 5-8

Die aus dem vorherigen Kapitel bereits bekannte Notation für Komponenten wird nun dahingehend erweitert, dass für *RequiredServiceReferenceSets* nun zusätzlich eine Schnittstellenrolle angegeben werden kann. Hierbei wird zunächst der Name des *RequiredServiceReferenceSets* angegeben. Gefolgt von einem Schrägstrich kann dann eine Schnittstellenrolle angegeben werden. Um schließlich die referenzierte Domänenschnittstelle nochmals hervorzuheben, kann durch einen Doppelpunkt getrennt diese Schnittstelle angegeben werden. In Abbildung 5-3 ist die bereits bekannte Sportlerkomponente *tim* dargestellt. Hierbei wurden die beiden *RequiredServiceReferenceSets* jeweils erweitert um eine Referenz auf eine Schnittstellenrolle.

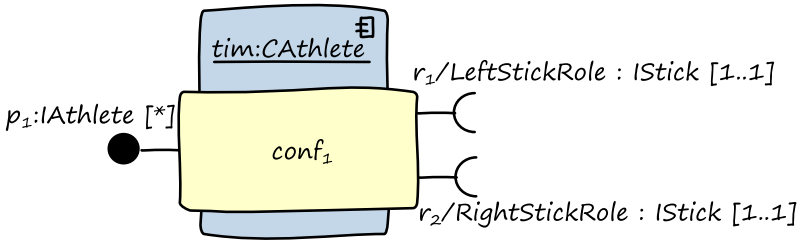


Abbildung 5-3: Athletenkomponente mit zwei *RequiredServiceReferenceSets* für linken und rechten Skistock

Für *ProvidedServices* ändert sich bzgl. der Notation nichts. Im nun folgenden Abschnitt wird gezeigt, auf welche Weise das Framework sicherstellt, dass *RequiredServiceReferenceSets* nur mit denjenigen *ProvidedServices* verbunden werden, welche konform zu einer ggf. referenzierten Schnittstellenrolle sind. Im Anschluss daran wird anhand eines Beispiels der resultierende Konfigurationsprozess beschrieben.

5.4 Berücksichtigung von Schnittstellenrollen im Konfigurationsprozess

Der Konfigurationsprozess, welcher im vorherigen Kapitel vorgestellt wurde, muss nun derart angepasst werden, dass bei der Auflösung von Abhängigkeiten die Konformität zu ggf. referenzierten Schnittstellenrollen berücksichtigt wird.

Die Grundidee für die Integration des Rollenkonzeptes in den Konfigurationsprozess basiert darauf, bei der Zuteilung eines Dienstes zu einem *RequiredServiceReferenceSet* zunächst zu prüfen, ob das *RequiredServiceReferenceSet* eine Schnittstellenrolle referenziert. Ist dies der Fall, wird zusätzlich geprüft, ob der Dienst die geforderten Eigenschaften der Rolle besitzt und ruft hierzu das Prädikat *isConformTo* der referenzierten Rolle auf. Liefert das Prädikat *true* zurück, erfolgt die Zuteilung des Dienstes, andernfalls nicht.

Da sich der Zustand eines Dienstes und damit auch deren Rollenzugehörigkeit jederzeit ändern kann, muss zyklisch überprüft werden, ob ein Dienst noch die Anforderungen erfüllt, die durch eine Schnittstellenrolle gegeben sind. Falls nicht, so muss der Dienst dem entsprechenden *RequiredServiceReferenceSet* durch das Framework wieder entzogen werden.

Der im vorherigen Kapitel vorgestellte Konfigurationsmechanismus muss lediglich auf Seiten des *ProvidedService* angepasst werden. Änderungen sind hierbei bei der Reaktion auf die Trigger *wantsUse*, *wantsNotUse* sowie beim Eintritt in den Zustand RUNNING nötig. Zusätzlich muss zyklisch die Zugehörigkeit eines Dienstes zur jeweils angefragten Rolle überprüft werden,

unabhängig ob ein Trigger aufgerufen wurde oder nicht. Diese einzelnen Anpassungen werden nun in den folgenden Abschnitten beschrieben.

5.4.1 Anpassung der Reaktion bei Eintritt in den Zustand RUNNING

Beim Eintritt in den Zustand RUNNING wird der Dienst durch das Framework nutzungsinteressierten *RequiredServiceReferenceSets* zugeordnet. Hierbei wird denjenigen Elementen in der Liste *runRequestedBy* jeweils der Dienst durch Aufruf von *serviceAssigned* zugeordnet, bis die maximale Anzahl an Nutzern für diesen Dienst erreicht ist. Dies wurde durch die Methode *assignServiceInitially* umgesetzt, welche in Kapitel 4 vorgestellt wurde und in Listing 5-2 nochmals dargestellt ist.

```
1  assignServiceInitially()  {
2   for(RequiredServiceReferenceSet rsrs :
3                          this.runRequestedBy.subset(0, maxNoOfUsers)) {
4    rsrs.assignedTo(this);
5    this.assignedTo.add(rsrs);
6   }
7  }
```

Listing 5-2: Die ursprüngliche Methode *assignServiceInitially*

Um nun die Konformität zu einer ggf. durch ein *RequiredServiceReferenceSet* referenzierten Rolle zu gewährleisten, muss diese Konformität vor der Zuweisung überprüft werden. Hierzu wird somit für jeden *ProvidedService* der Menge *runRequestedBy* zunächst die Methode *isConformTo* aufgerufen. Erst wenn diese *true* zurückliefert, kann eine Zuordnung des Dienstes erfolgen. Liefert die Methode hingegen *false* zurück, so kann der Dienst vorerst nicht dem betreffenden *RequiredServiceReferenceSet* zugeordnet werden. Jedoch überprüft das Framework zyklisch die Konformität für den Fall, dass zu einem späteren Zeitpunkt die Konformität gegeben ist (siehe 5.4.4). Referenziert ein *RequiredServiceReferenceSet* hingegen keine Schnittstellenrolle, so kann die Überprüfung entfallen. In diesem Fall ist lediglich die Übereinstimmung der Domänenschnittstelle einziges Kriterium.

Da die Überprüfung der Rollenkonformität nur dann stattfinden kann, wenn der Dienst ausgeführt wird, kommt eine Überprüfung ausschließlich im Zustand RUNNING in Frage. Sämtliche Anpassungen beziehen sich somit auf *ProvidedServices* im Zustand RUNNING bzw. beim Übergang in diesen Zustand. Die angepasste Version der Methode *assignServiceInitially* ist in Listing 5-3 dargestellt, wobei die hinzugekommenen Zeilen fett gedruckt ist.

```
1  assignServiceInitially()  {
2   for(RequiredServiceReferenceSet rsrs : this.runRequestedBy) {
3    if(rsrs.requires == null || rsrs.requires.isConformTo(this)) {
4     rsrs.assignedTo(this);
5     this.assignedTo.add(rsrs);
6     if(this.assignedTo.size() == this.maxNoOfUsers) { break; }
7    }
8   }
9  }
```

Listing 5-3: Die neue Methode *assignServiceInitially*, welche die Rollenzugehörigkeit des Dienstes berücksichtigt

Auf diese Weise ist sichergestellt, dass ein Dienst, welcher in den Zustand RUNNING wechselt, nur denjenigen *RequiredServiceReferenceSets* zugeteilt wird, deren referenzierte Rolle auch durch den Dienst realisiert wird.

5.4.2 Anpassung der Reaktion bei Aufruf des Triggers *wantsUse*

Befindet sich ein *ProvidedService* im Zustand RUNNING und kommt ein neuer Nutzungsinteressent durch Aufruf von *wantsUse* hinzu, erfolgte bislang ebenfalls eine Zuordnung des Dienstes an das *RequiredServiceReferenceSet*, sofern die Obergrenze für die Anzahl der gleichzeitigen Nutzer noch nicht erreicht war. In Kapitel 4 wurde diese Funktionalität in der Methode *assignService* umgesetzt, welche in Listing 5-4 nochmals dargestellt ist.

```
1  assignService(RequiredServiceReferenceSet rsrs)  {
2   if(this.assignedTo.size()<self.maxNoOfUsers) {
3    rsrs.serviceAssigned(this);
4    this.assignedTo.add(rsrs);
5   }
6  }
```

Listing 5-4: Die ursprüngliche Methode *assignService*

Die dort enthaltene Bedingung wird nun erweitert um die Überprüfung der Rollenkonformität. Listing 5-5 zeigt die angepasste Version der Methode *assignService*, wobei die Änderungen wiederum fett gedruckt sind:

```
1  assignService(RequiredServiceReferenceSet rsrs)  {
2   if(this.assignedTo.size()<self.maxNoOfUsers) {
3    if(rsrs.requires == null || rsrs.requires.isConformTo(this)) {
4     rsrs.assignedTo(this);
5     this.assignedTo.add(rsrs);
6    }
7   }
8  }
```

Listing 5-5: Die neue Methode *assignService*, welche die Rollenzugehörigkeit des Dienstes berücksichtigt

Auch hier wird durch die Änderung bewirkt, dass der Dienst nur noch denjenigen *RequiredServiceReferenceSets* zugeordnet wird, deren ggf. referenzierte Rolle durch den Dienst realisiert wird.

5.4.3 Anpassung der Reaktion bei Aufruf des Triggers *wantsNotUse*

Zieht nun ein *RequiredServiceReferenceSet* das Nutzungsinteresse durch Aufruf des Triggers *wantsNotUse* zurück und befindet sich das *RequiredServiceReferenceSet* in der Menge *assignedTo*, so kann der Dienst nun ggf. anderen *RequiredServiceReferenceSets* der Menge *runRequestedBy* zugeteilt werden. Dieser Fall tritt insbesondere dann ein, wenn zuvor die maximale Anzahl an Nutzern erreicht war und noch weitere nutzungsinteressierte *RequiredServiceReferenceSets* vorliegen. Bislang wurde diese Dienstneuzuteilung durch die Methode *replaceServiceUser* aus Kapitel 4 vorgenommen, welche in Listing 5-6 nochmals abgebildet ist.

```
1  replaceServiceUser(RequiredServiceReferenceSet oldUser)  {
2   this.assignedTo.remove(oldUser);
3   for(RequiredServiceReferenceSet rsrs : this.runRequestedBy) {
4    if(!this.assignedTo.contains(rsrs)) {
5     rsrs.serviceAssigned(this);
6     this.assignedTo.add(rsrs);
7     return;
8    }
9   }
10 }
```

Listing 5-6: Die ursprüngliche Methode *replaceServiceUser*

Soll nun die Rollenkonformität berücksichtigt werden, muss vor einer Neuzuordnung zunächst geprüft werden, ob der neu zuzuordnende Dienst alle geforderten Eigenschaften der referenzierten Rolle erfüllt. Die Methode *replaceServiceUser* wird deshalb um diese Abfrage erweitert. Die neue Version dieser Methode ist in Listing 5-7 angegeben.

```
1  replaceServiceUser(RequiredServiceReferenceSet oldUser)  {
2   this.assignedTo.remove(oldUser);
3   for(RequiredServiceReferenceSet rsrs : this.runRequestedBy) {
4    if(!this.assignedTo.contains(rsrs)) {
5     if(rsrs.requires == null || rsrs.requires.isConformTo(this)) {
6      rsrs.assignedTo(this);
7      this.assignedTo.add(rsrs);
8      return;
9     }
10   }
11  }
12 }
```

Listing 5-7: Die neue Methode *replaceServiceUser*, welche die Rollenzugehörigkeit des Dienstes berücksichtigt.

Wie gezeigt wurde, findet eine Überprüfung der Rollenzugehörigkeit ausschließlich dann statt, wenn sich der Dienst im Zustand RUNNING befindet. Der Grund hierfür ist der, dass erst dann sichergestellt ist, dass das Prädikat auch auf valide Werte zurückgreifen kann.

5.4.4 Zyklische Überprüfung der Rollenzugehörigkeit

Da sich der Zustand eines Dienstes und damit auch seine Rollenzugehörigkeit jederzeit ändern können, muss das Framework zyklisch die Rollenkonformität überprüfen. Dies gilt sowohl für Dienste, welche bereits in Verwendung sind, als auch für Dienste, die bislang auf Grund mangelnder Konformität nicht verwendet werden. Eine zyklisch Überprüfung muss für all jene *RequiredServiceReferenceSets* durchgeführt werden, welche Elemente der Menge *runRequestedBy* eines oder mehrerer Dienste sind.

Eine Reaktion muss in zwei Fällen erfolgen. Falls ein Dienst einem *RequiredServiceReferenceSet* zugeteilt wurde, und der Dienst diese Rolle zum Zeitpunkt der Überprüfung nicht mehr erfüllt, so muss dem *RequiredServiceReferenceSet* der Dienst entzogen werden. Falls auf der anderen Seite zum Zeitpunkt der Überprüfung der Dienst auf Grund einer Zustandsänderung nun doch einem Nutzungsinteressierten zugewiesen kann, so muss eben diese Zuordnung erfolgen. Der Zustand RUNNING von *ProvidedService* wird zu diesem Zweck um eine Methode *checkRoleConformance* erweitert, die durch das Framework zyklisch ausgeführt wird. Der erweiterte Zustand RUNNING von *ProvidedService* ist in Abbildung 5-4 dargestellt.

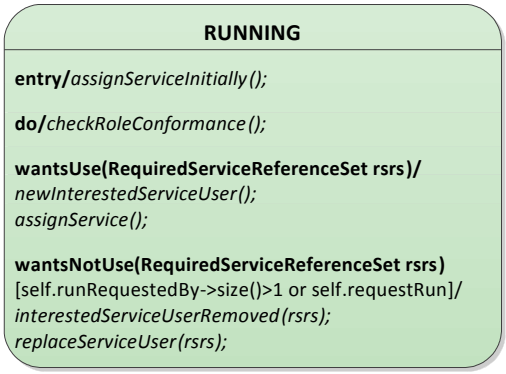


Abbildung 5-4: Der Zustand RUNNING von *ProvidedService* erweitert um die regelmäßige Rollenüberprüfung

Die Methode *checkRoleConformance* ist in Listing 5-8 abgebildet.

```
1  checkRoleConformance()  {
2   for(RequiredServiceReferenceSet rsrs : this.assignedTo) {
3    if(rsrs.requires != null && (!rsrs.requires.isConformTo(this)) {
4     rsrs.serviceNotAssigned(this);
5    }
6   }
7
8   for(RequiredServiceReferenceSet rsrs : this.runRequestedBy) {
9    if(!this.assignedTo.contains(rsrs) &&
10      rsrs.requires.isConformTo(this) &&
11      this.assignedTo.size() < this.maxNoOfUsers) {
12       rsrs.serviceAssigned(this);
13    }
14   }
15  }
   }
```

Listing 5-8: Die Methode *checkRoleConformance*

Die Methode *checkRoleConformance* iteriert zunächst über all diejenigen *RequiredServiceReferenceSets*, denen der Dienst zugeteilt wurde, also Elemente der Menge *assignedTo* sind. Ist ein zugeteilter Dienst nicht mehr konform zur Schnittstellenrolle, welche vom *RequiredServiceReferenceSet* referenziert wird, so wird ihm der Dienst entzogen.

Im zweiten Teil der Methode wird dann für diejenigen *RequiredServiceReferenceSets* die Rollenkonformität des Dienstes überprüft, welche den Dienst zum Zeitpunkt der Überprüfung noch nicht nutzen. Ist der Dienst dann zu einem oder mehreren *RequiredServiceReferenceSets* kompatibel, wird er ihnen zugewiesen, solange die maximale Anzahl an Nutzern noch nicht erreicht ist.

Mit Hilfe dieser Änderungen am ursprünglichen Konfigurationsprozess ist es nun möglich, Dienste nur denjenigen *RequiredServiceReferenceSets* zuzuordnen, bei denen eine Rollenkonformität gegeben ist. Im nun folgenden Abschnitt werden nun die Auswirkungen der Änderungen am Komponentenmodell anhand einiger Beispiele illustriert.

5.5 Beispiel eines Konfigurationsprozesses unter Berücksichtigung von Schnittstellenrollen

Grundlage der nun folgenden Beispiele bildet die bereits bekannte Sportlerkomponente aus Abbildung 5-3. Sie besitzt eine Komponentenkonfiguration mit zwei *RequiredServiceReferenceSets*, wobei beide auf die Domänenschnittstelle *IStick* verweisen. Sie unterscheiden sich lediglich in der referenzierten *InterfaceRole*. Das Prädikat der jeweiligen Rolle sei wie in Formel 5-9 angegeben definiert:

$$\textbf{context}\ LeftStickRole :: isConformTo(IStick\ ps): bool$$
$$\textbf{post}: result = ps.getSide() = 'left'$$

$$\textbf{context}\ RightStickRole :: isConformTo(IStick\ ps): bool$$
$$\textbf{post}: result = ps.getSide() = 'right'$$

Formel 5-9

Da die ersten Schritte des Konfigurationsprozesses gegenüber dem vorgestellten Verfahren aus Kapitel 4 unverändert sind, werden diese hier nicht nochmals erläutert. Die Ausgangssituation für die folgenden Beispielabläufe ist in Abbildung 5-5 dargestellt.

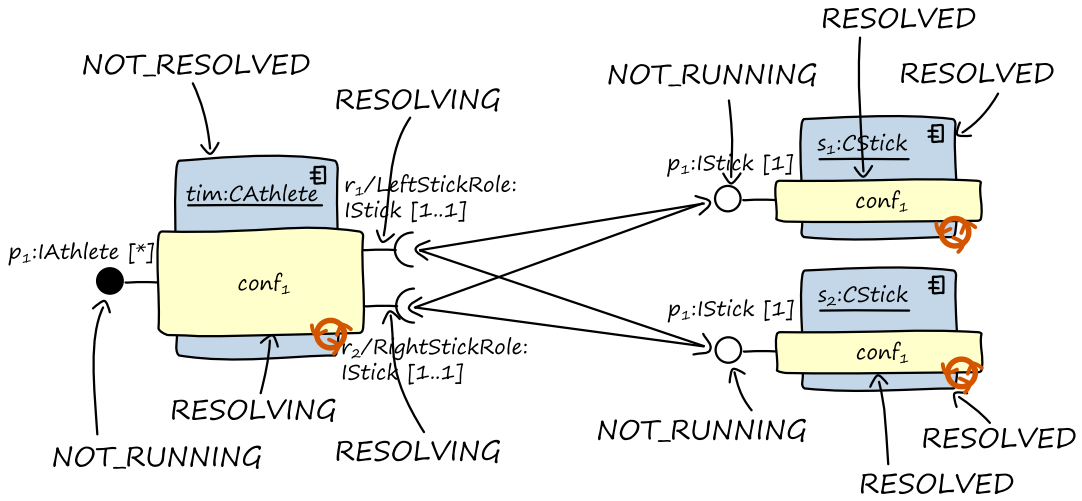


Abbildung 5-5: Ausgangssituation für die Beispiele zur Rollenkompatibilität

Die Abbildung zeigt die Athletenkomponente, und zudem zwei Skistockkomponenten mit jeweils einer *ComponentConfiguration* und einem *ProvidedService*. Die dargestellte Situation ist in folgenden Schritten entstanden. Der *ProvidedService* der Sportlerkomponente hat zunächst der *ComponentConfiguration* $conf_1$ auf Grund des gesetzten *requestRun*-Flags gemeldet, dass er ausgeführt werden möchte. Daraufhin hat die Konfiguration beiden *RequiredServiceReferenceSets* mitgeteilt, dass sie ihre Abhängigkeiten zu Diensten anderer Komponenten auflösen sollen. Diese haben daraufhin jeweils eine Nutzungsanfrage an beide *ProvidedServices* gestellt. Daraufhin wurde ihre jeweilige *ComponentConfiguration* über den Ausführungswunsch informiert. Da keine der Skistockkomponenten *RequiredServiceReferenceSets* definieren, konnten die *ComponentConfigurations* direkt in den Zustand RESOLVED wechseln. Darüber wurde dann ihre jeweilige *DynamicAdaptiveComponent* informiert. Abbildung 5-5 zeigt das Resultat dieser durchgeführten Aktionen.

Im nun folgenden Schritt teilen die Komponenten s_1 und s_2 ihrem jeweils angebotenen Dienst durch Aufruf des Triggers *serviceRunnable* mit, dass sie ausgeführt werden können. Da für beide Dienste Nutzungsanforderungen vorliegen, erfolgt nun der Wechsel in den Zustand RUNNING und somit die anschließende Zuweisung des Dienstes an die Nutzer. Erst an dieser Stelle des Konfigurationsprozesses kann festgestellt werden, ob die Dienste zu einer der Schnittstellenrollen kompatibel sind, und wenn ja zu welcher.

Für dieses Beispiel sei angenommen, dass der Dienst p_1 der oberen Skistockkomponente aus Abbildung 5-5 bei Aufruf von *getSide* den Wert „left“ zurückliefert, und p_1 der unteren Skistockkomponente den Wert „right“. Beide Dienste führen unmittelbar nach Eintritt in den Zustand RUNNING die Methode *assignServiceInitially* aus Listing 5-3 aus. Daraufhin wird festgestellt, dass für die obere Skistockkomponente lediglich das *RequiredServiceReferenceSet* r_1 als Nutzer in Frage kommt. Entsprechend wird der Dienst durch Aufruf von *serviceAssigned* zugewiesen. Der andere Dienst wird dem *RequiredServiceReferenceSet* r_2 zugeordnet. Das Resultat dieser Aktionen ist in Abbildung 5-6 dargestellt.

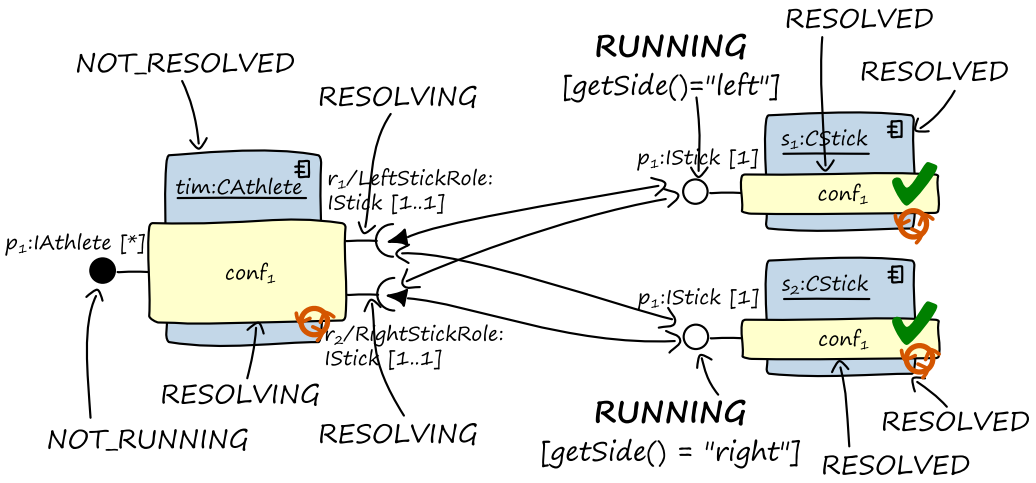


Abbildung 5-6: Situation nach der Erstzuordnung der Dienste zum jeweils geeigneten *RequiredServiceReferenceSet*

Die *RequiredServiceReferenceSets* können anschließend in den Zustand RESOLVED wechseln und teilen dies der *ComponentConfiguration* $conf_1$ mit. Diese kann dann ebenfalls in den Zustand RESOLVED wechseln und übermittelt diese Information durch Aufruf des Triggers *configurationResolved* der *DynamicAdaptiveComponent*. Diese teilt dem Dienst p_1 über den Trigger *serviceRunnable* mit, dass er ausgeführt werden kann. Der Dienst wechselt daraufhin in den Zustand RUNNING. Abbildung 5-7 zeigt das Resultat des Konfigurationsprozesses.

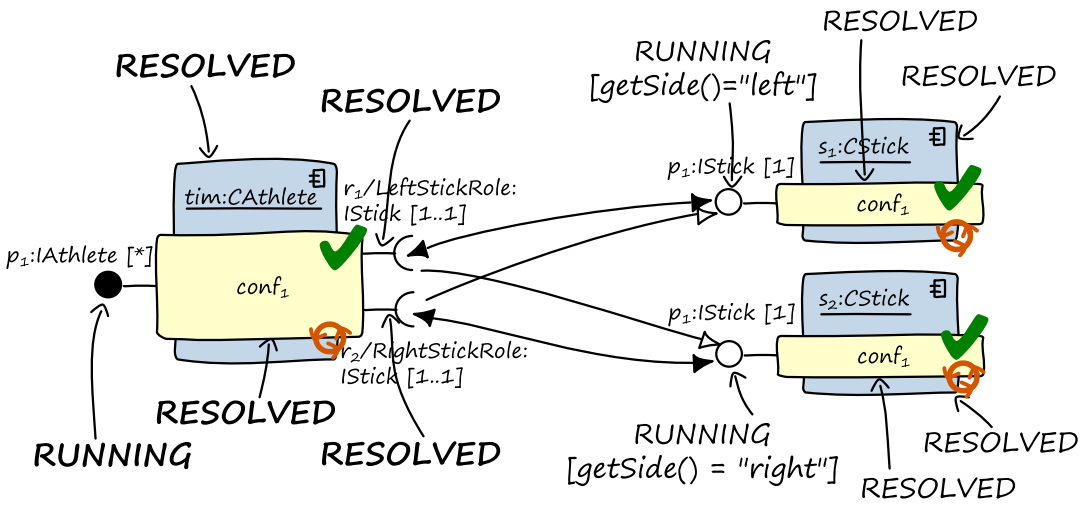


Abbildung 5-7: Resultat des Konfigurationsprozesses

Wie die Abbildung ebenfalls zeigt, hat das *RequiredServiceReferenceSet* r_1 seine Nutzungsanforderung an den Dienst p_1 der unteren Skistockkomponente zurückgezogen. Ebenso hat r_2 den Nutzungswunsch für p_1 der unteren Skistockkomponente zurückgezogen. Der Grund hierfür ist der, dass beiden *RequiredServiceReferenceSets* die maximale Anzahl an benötigten Diensten zur Verfügung stehen. Sobald allerdings einer der Skistockdienste die Nutzungsrechte z.B. mangels Konformität zur jeweils angeforderten Rolle entzieht, stellt das jeweilige *RequiredServiceReferenceSet* erneut Nutzungsanfragen an alle verfügbaren Dienste, die die Domänenschnittstelle *IStick* implementieren. Der Vorteil dieses Vorgehens ist, dass Dienste nicht unnötigerweise ausgeführt werden. Bei häufigem Wechsel der Zuordnung von *ProvidedService* zu *InterfaceRole* kann dieses Vorgehen allerdings zu umfangreichen Rekonfigurations-Maßnahmen führen.

Zum Abschluss des Beispiels sei nun noch angenommen, dass der Dienst p_1 der oberen Skistockkomponente aus Abbildung 5-7 nicht mehr den Wert „left“ zurückliefert, sondern den Wert „right“. Sobald nun eine Überprüfung durch Aufruf der Methode *checkRoleConformance* aus Listing 5-8 durchgeführt wird, wird dieser Umstand erkannt. Dem *RequiredServiceReferenceSet* r_1 wird daraufhin der Dienst entzogen, woraufhin die Athletenkomponente und letztlich dessen *ProvidedService* nicht mehr ausgeführt werden kann. Das *RequiredServiceReferenceSet* r_1 stellt nun wieder Nutzungsanfragen an beide verfügbaren Dienste. Somit prüfen beide Dienste regelmäßig, ob sie zu r_1 passen. Liefert der Dienst p_1 der oberen Skistockkomponente nun später wieder „left“ zurück, so wird der angebotene Dienst der Athletenkomponente nach wenigen Konfigurationsschritten wieder ausgeführt.

5.6 Zusammenfassung

In diesem Kapitel wurde das Komponentenmodell aus Kapitel 4 dahingehend erweitert, dass die Möglichkeiten zur Spezifikation von Anforderungen an benötigte Dienste mächtiger geworden sind. Hierzu wurde das Konzept der Schnittstellenrollen eingeführt. Unter Verwendung von Schnittstellenrollen können Anforderungen spezifiziert werden, welche sich auf den Rückgabewert von Domänenschnittstellen-Methoden beziehen. Zudem wurde das Framework dahingehend erweitert, dass es nun die Rollenkonformität von *ProvidedServices* bei der Verschaltung berücksichtigt.

Die vorgenommenen Anpassungen haben die wesentlichen Eigenschaften der behandelten Systeme nicht verändert. Die Konfiguration basiert weiterhin auf selbstorganisierenden Komponenten, und das Framework kommt auch weiterhin ohne zentralen Konfigurator aus.

Im nun folgenden Kapitel wird das erweiterte Komponentenmodell nochmals wesentlich ergänzt. Es wird dann die Möglichkeit geben, Anforderungen an das Gesamtsystem spezifizieren zu können. Das Ergebnis ist dann ein Framework, welches die automatische Konfiguration verteilter mobiler Anwendungen auf Basis selbstorganisierender Komponenten ermöglicht, und zusätzlich komponentenübergreifende Anforderungen berücksichtigt.

6 Anwendungsarchitektur-konforme Konfiguration

In den vorangegangenen Kapiteln wurde ein Framework vorgestellt, welches die automatische Konfiguration dynamisch-adaptiver Systeme auf Basis selbstorganisierender Komponenten ermöglicht. Wurde eine selbstorganisierende Komponente installiert, so hat das Framework unter bestimmten Voraussetzungen automatisch damit begonnen, Abhängigkeiten zu Diensten anderer Komponenten aufzulösen. Die Voraussetzung war, dass die Komponente einen Dienst mit gesetztem *requestRun*-Flag anbietet oder ein potentieller Nutzer eines angebotenen Dienstes im System installiert ist.

Anforderungen an Dienste anderer Komponenten konnten mit Hilfe von *RequiredServiceReferenceSets* spezifiziert werden. Hierbei wurden die Möglichkeiten zur Anforderungsspezifikation in Kapitel 5 dahingehend erweitert, dass auch Rückgabewerte von Domänenschnittstellenmethoden berücksichtigt werden konnten.

Die bislang betrachteten Systeme haben eines gemeinsam: Sie entstehen durch das selbständige Auflösen von Abhängigkeiten von Komponenten zu anderen Komponenten sowie dem selbständigen Starten von Diensten, für den Fall, dass eine Ausführung erforderlich ist. Im Rahmen des Konfigurationsprozesses wurden allerdings komponentenübergreifende, Anwendungsarchitektur-spezifische Anforderungen nicht berücksichtigt. Es sind lediglich komponentenlokale Anforderungen in den Konfigurationsprozess eingeflossen.

Wie im folgenden Abschnitt gezeigt wird, existieren allerdings häufig Anwendungsfälle, in denen es unerlässlich ist, den Konfigurationsprozess derart zu beeinflussen, dass komponentenübergreifende, anwendungsspezifische Anforderungen berücksichtig werden. Im weiteren Verlauf des Kapitels wird das bereits aus den vorherigen Kapiteln bekannte Framework abschließend dahingehend erweitert, dass zum einen die Spezifikation komponentenübergreifender Anforderungen möglich wird, und zum anderen, dass das Framework basierend auf selbstorganisierenden Komponenten diese Anforderungen automatisiert umsetzt.

6.1 Motivation

In diesem Abschnitt wird anhand eines kleinen Beispiels gezeigt, weshalb die Einhaltung komponentenübergreifender Anforderungen in einigen Fällen unerlässlich ist. Anschließend wird die Lösungsidee skizziert, bevor in den folgenden Abschnitten die Lösung schrittweise vorgestellt wird.

6.1.1 Problemstellung

Für das Beispiel sei angenommen, dass ein selbstorganisierendes System entwickelt werden soll, welches das Training von Biathleten unterstützt, so wie es bereits in Kapitel 3 umrissen wurde. Das System soll im konkreten Fall die im Folgenden beschriebenen Dienste erbringen.

Zunächst soll einem Trainer eine Übersicht über die Leistungsdaten all seiner Athleten angezeigt werden, wobei mindestens die Daten eines Athleten angezeigt werden sollen. Dazu sei angenommen, dass die in Abbildung 6-1 dargestellte Komponente zur Verfügung steht.

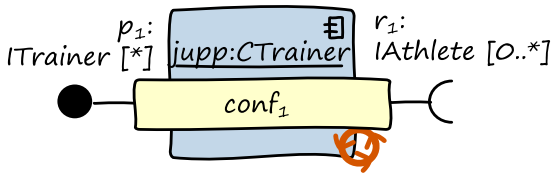


Abbildung 6-1: Die für das System zur Verfügung stehende Trainerkomponente

Die geforderte Funktionalität wird über den Dienst p_1 bereitgestellt, welcher die Schnittstelle *ITrainer* implementiert. Der Dienst definiert eine Abhängigkeit zu Diensten, welche die Schnittstelle *IAthlete* implementieren. Allerdings kann der Dienst auch dann ausgeführt werden, wenn kein Athletendienst im System verfügbar ist. Um der Anforderung gerecht zu werden, dass der Trainerdienst erst dann ausgeführt werden soll, wenn er Zugriff auf mindestens einen Athletendienst hat, müsste die Implementierung der Trainerkomponente angepasst werden. Zudem müsste das Attribut *minNoOfRequiredRefs* von r_1 aus Abbildung 6-1 auf 1 gesetzt werden. Doch nicht immer kann der Code einer Komponente derart angepasst werden. Zudem widerspricht es der ursprünglichen Zielsetzung einer Komponente, sie für jeden konkreten Einsatzzweck manuell anzupassen. Die im weiteren Verlauf dieses Kapitels vorgestellte Lösung erlaubt die anwendungsspezifische Spezifikation der Anzahl minimal bzw. maximal benötigter Referenzen für *RequiredServiceReferenceSets*, ohne den Quellcode der Komponente anpassen zu müssen.

Die Leistungsdaten einzelner Athleten innerhalb der Anwendung über die Schnittstelle *IAthlete* angeboten werden. Für das Beispiel sei angenommen, dass die in Abbildung 6-2 dargestellte Komponente zur Verfügung steht.

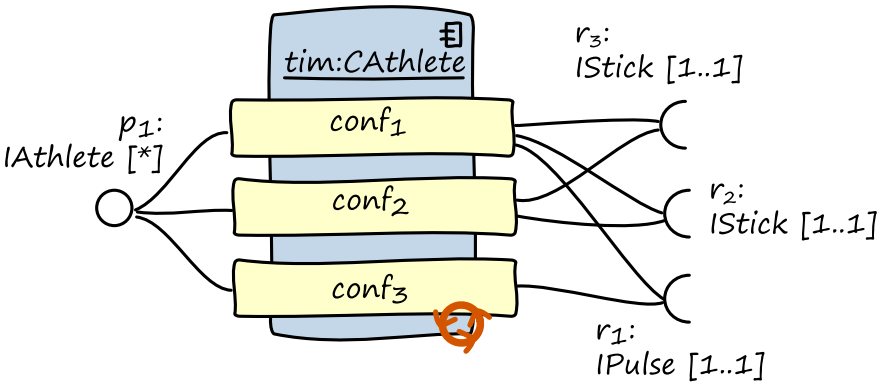


Abbildung 6-2: Die für das System zur Verfügung stehende Athletenkomponente

Die Komponente definiert drei *ComponentConfigurations*, wobei $conf_1$ als beste Konfiguration und $conf_3$ als schlechteste Konfiguration spezifiziert wurde. Die Konfiguration $conf_3$ kann dann aktiviert werden, falls r_1 mit einem Dienst verbunden werden kann, der die Schnittstelle *IPulse* implementiert. Die Konfiguration $conf_2$ kann dann aktiviert werden, wenn r_2 und r_3 mit jeweils einem Skistockdienst verbunden sind. Und die Konfiguration $conf_1$ wird dann aktiviert, falls die Abhängigkeiten aller drei *RequiredServiceReferenceSets* aufgelöst werden konnten. In allen drei Konfigurationen bietet die Komponente einen Dienst an, der die Domänenschnittstelle *IAthlete* implementiert. Diese definiert zum einen eine Methode *getPulse():int* zum Abfragen des aktuellen Puls, sowie eine Methode *getSkiingTechnique():String*, welche die aktuell verwendete Lauftechnik (Doppelstockschub/Diagonalschritt) zurückliefert. Ist die Konfiguration $conf_3$ aktiv,

so liefert der Aufruf von *getSkiingTechnique* den Wert *null* zurück. Ist hingegen die Konfiguration $conf_2$ aktiv, so liefert der Aufruf von *getPulse* den Wert *-1* zurück.

Für die Beispielanwendung wird nun angenommen, dass insbesondere die Lauftechnik analysiert werden soll. D.h. für die Anwendung ist ausschließlich die *ComponentConfiguration* $conf_2$ der Athletenkomponente *tim* aus Abbildung 6-2 relevant. Selbst wenn also ein Pulsdienst sowie zwei Skistockdienste verfügbar sein sollten, soll für die Anwendung nicht die Konfiguration $conf_1$ aktiviert werden, obwohl dies laut Komponentenspezifikation die beste Konfiguration ist. Mit dem bislang vorgestellten Framework, welches in Kapitel 4 und 5 beschrieben wurde, besteht nicht die Möglichkeit, anwendungsspezifisch Einfluss auf die zu aktivierende *ComponentConfiguration* einer Komponente zu nehmen. Es bleibt auch hier bislang nur die Möglichkeit, die Implementierung der Komponente anwendungsspezifisch anzupassen. Im Verlaufe dieses Kapitels wird das bestehende Framework dahingehend erweitert, dass eine derartige, anwendungsspezifische Einflussnahme auf die Aktivierung von Komponentenkonfigurationen möglich ist.

Das hier beschriebene Beispielsystem soll abschließend noch das Schießtraining ermöglichen. Hierbei soll für jeden Athleten eine Schießbahn zur Verfügung stehen. Jede Schießbahn soll im System durch einen Dienst repräsentiert sein, der jeweils die Domänenschnittstelle *IShootingLine* implementiert. Ein Beispiel für eine solche Komponente ist in Abbildung 6-3 dargestellt.

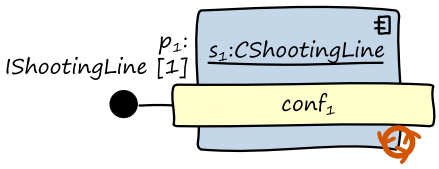


Abbildung 6-3: Die für das System zur Verfügung stehende Schießstandkomponente

Der Dienst p_1 der Komponente startet in diesem Fall auch dann, wenn keinerlei Verwender in Form einer anderen Komponente vorliegt, da das Flag *requestRun* gesetzt ist (durch den gefüllten Kreis symbolisiert). Für das Beispiel soll das System allerdings das Schießen erst dann ermöglichen, wenn eine Schießaufsicht anwesend ist. Diese wird im System durch einen Dienst repräsentiert, der die Domänenschnittstelle *ISupervisor* implementiert. Die in Abbildung 6-4 dargestellte Komponente bietet einen solchen Dienst an.

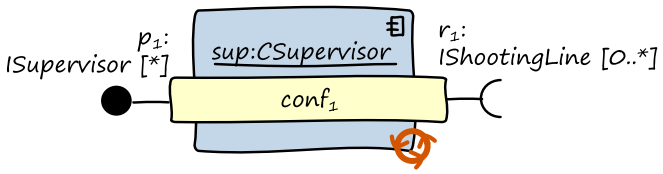


Abbildung 6-4: Die für das System zur Verfügung stehende Schießaufsichtkomponente

An dieser Stelle kommt die wohl komplexeste Anforderung zum Tragen, die an das System gestellt wird. Denn im System soll sichergestellt sein, dass für jeden Sportler, der mit der Trainerkomponente verbunden ist, genau eine Schießstandkomponente zur Verfügung steht. Das bedeutet, dass die Anzahl derjenigen Dienste, welche von der Schießaufsichtkomponente verwendet werden, mit der Anzahl derjenigen Athletenkomponenten übereinstimmen muss, auf die die Trainerkomponente zugreift.

Eine Systemkonfiguration, welche alle zuvor beschriebenen Anforderungen erfüllt, ist in Abbildung 6-5 dargestellt.

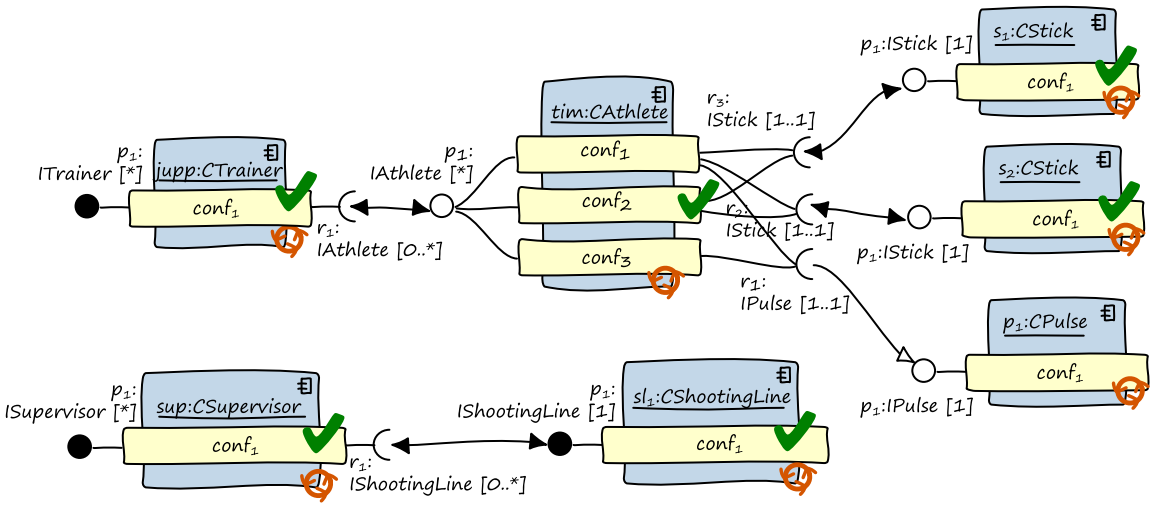


Abbildung 6-5: Eine Konfiguration, welche alle zuvor definierten Anforderungen erfüllt.

Hier ist eine Trainerkomponente mit einer Sportlerkomponente verbunden, die wiederum mit einem linken und einem rechten Skistock verbunden ist. Des Weiteren besteht die Anwendung aus einer Schießaufsichtkomponente, welche wiederum mit einer Schießbahnkomponente verbunden ist.

Im weiteren Verlauf des Kapitels wird nun das bestehende Framework erweitert um die Fähigkeit, Konfigurationen automatisiert erzeugen zu können, die derartige Anforderungen berücksichtigen. Kommt zum Beispiel in der oben angegebenen Konfiguration eine neue Sportlerkomponente in das System, so soll diese erst dann in die Anwendung integriert werden, sobald auch eine Schießbahnkomponente für diesen Sportler zur Verfügung steht. Und die Anwendung wird beispielsweise dann gestoppt, wenn die Sportlerkomponente aus Abbildung 6-5 nur noch mit einer Skistockkomponente verbunden ist.

Des Weiteren berücksichtigt das Framework Anforderungen, die insbesondere in emergenten Systemen häufig gefordert sind. So ermöglicht die hier vorgestellte Lösung die nachträgliche Integration neuer, zum Systemstart unbekannter Komponenten. Das Framework ist in der Lage zu beurteilen, ob und wie diese Komponenten in die laufende Anwendung integriert werden können und kann diese Integration automatisch durchführen.

Im folgenden Abschnitt wird nun ein kurzer Überblick über den Lösungsansatz gegeben, bevor dann in den weiteren Abschnitten eine detaillierte Vorstellung der Lösung folgt.

6.1.2 Lösungsskizze

Ziel des im Rahmen dieser Arbeit entwickelten Frameworks ist es, basierend auf einer Menge zur Verfügung stehender Komponenten eine Anwendungskonfiguration zu erzeugen, welche alle spezifizierten Anforderungen an die Anwendungsarchitektur erfüllt. Eine solche Anwendungskonfiguration besteht jeweils aus einer Menge von Komponenten sowie aus Verbindungen zwischen diesen Komponenten. Aufgabe des Frameworks ist es somit zunächst, aus der Menge aller zur Verfügung stehenden Komponenten diejenigen auszuwählen, welche für eine Anwendungsarchitektur-konforme Konfiguration in Frage kommen. Zusätzlich müssen die

Komponenten dann derart miteinander verbunden werden, dass alle spezifizierten Anforderungen erfüllt sind.

Die im Folgenden vorgestellte Lösung ermöglicht zum einen die Spezifikation von Bedingungen, unter denen Komponenten für die Verwendung innerhalb einer Anwendung in Betracht kommen. Und zum anderen kann definiert werden, wie diese Komponenten miteinander vernetzt werden sollen. Das entwickelte Framework ist später in der Lage, basierend auf einer solchen Spezifikation und einer Menge zur Verfügung stehender Komponenten eine Anwendungsarchitektur-konforme Konfiguration zu erzeugen. Außerdem konfiguriert das Framework die Anwendung automatisch um, sollten die Anforderungen nicht mehr erfüllt sein. Der Lösungsansatz basiert hierbei weiterhin auf einem System selbstorganisierender Komponenten, jedoch findet die Konfiguration hier mittels Unterstützung einer zentralen, aber leichtgewichtigen Konfigurationseinheit statt.

Die Kriterien zur Auswahl geeigneter Komponenten für eine Anwendung werden mit Hilfe sogenannter *Templates* definiert. Hierbei handelt es sich um Grundrisse bzw. Schablonen von Komponenten. Eine Anwendungsspezifikation besteht aus einer oder mehreren solcher *Templates*. So könnte die zuvor skizzierte Biathlonanwendung beispielsweise aus einem *Template* für Trainerkomponenten, einem für Sportlerkomponenten, einem für Schießbahnkomponenten usw. bestehen. Für jedes der *Templates* können Anforderungen hinterlegt werden, die angeben, unter welchen Umständen eine Komponente zu einem *Template* kompatibel ist. So kann z.B. für ein Athleten-*Template* hinterlegt werden, dass hier nur solche Komponenten kompatibel sind, welche einen Dienst anbieten, der die Domänenschnittstelle *IAthlete* implementiert. Zur Laufzeit sorgt das Framework dann später dafür, dass diesen *Templates* ausschließlich jeweils zur Schablone passende Komponenten zugeordnet werden. Im weiteren Verlauf der Arbeit wird ein *Template* als gestricheltes Rechteck dargestellt. Mittels gestrichelter Kreise und Halbkreise werden Anforderungen hinsichtlich angebotener und benötigter Dienste von Komponenten visualisiert (hierzu später mehr). In Abbildung 6-6 sind für eine Anwendung zwei *Templates* definiert, denen jeweils eine bzw. zwei Komponenten zugeordnet sind.

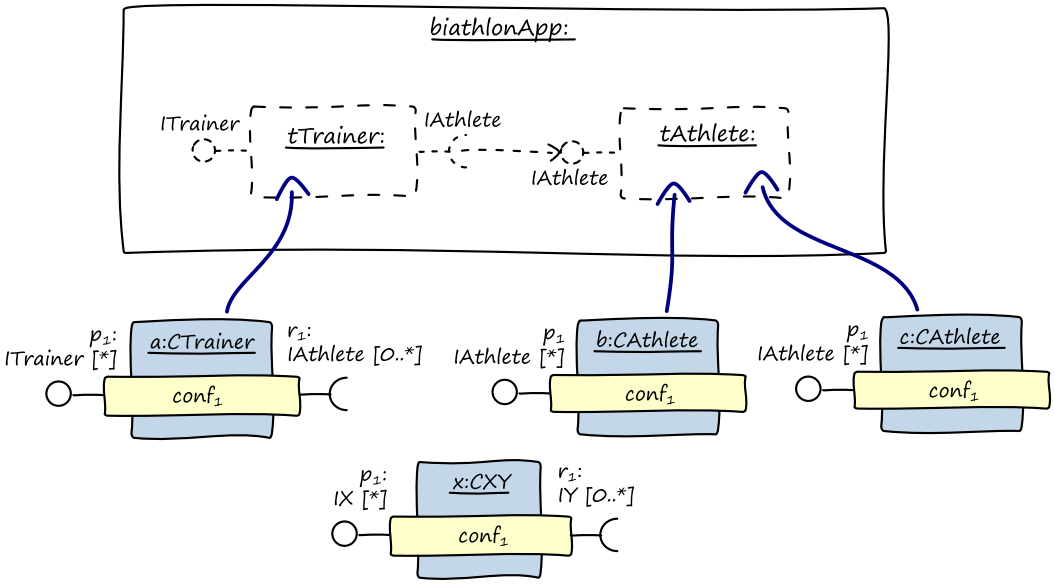


Abbildung 6-6: Auswahl geeigneter Komponenten für eine Anwendungskonfiguration

In diesem Beispiel bleibt eine Komponente für die Anwendung unberücksichtigt, da diese zu keinem *Template* der Anwendung kompatibel ist.

Im nächsten Schritt müssen nun die ausgewählten Komponenten miteinander verbunden werden, um ein lauffähiges System zu erhalten. Hierzu werden im Anwendungsgrundriss neben *Templates* auch Verbindungen zwischen *Templates* definiert, welche als gestrichelte Pfeile dargestellt werden (siehe Abbildung 6-6). Diese treffen Aussagen darüber, wie die zugeordneten Komponenten miteinander verbunden werden sollen. So kann beispielsweise definiert werden, dass jede Komponente, welche dem *Template tTrainer* aus Abbildung 6-6 zugeordnet wurde, mit mindestens einer Komponente verbunden sein muss, die dem *Template tAthlete* zugeordnet wurde. Das Framework sorgt später zur Laufzeit der Anwendung dafür, dass auch die Anforderungen, welche die Verbindungen zwischen Komponenten betreffen, berücksichtigt werden. Abbildung 6-7 zeigt eine mögliche resultierende Konfiguration.

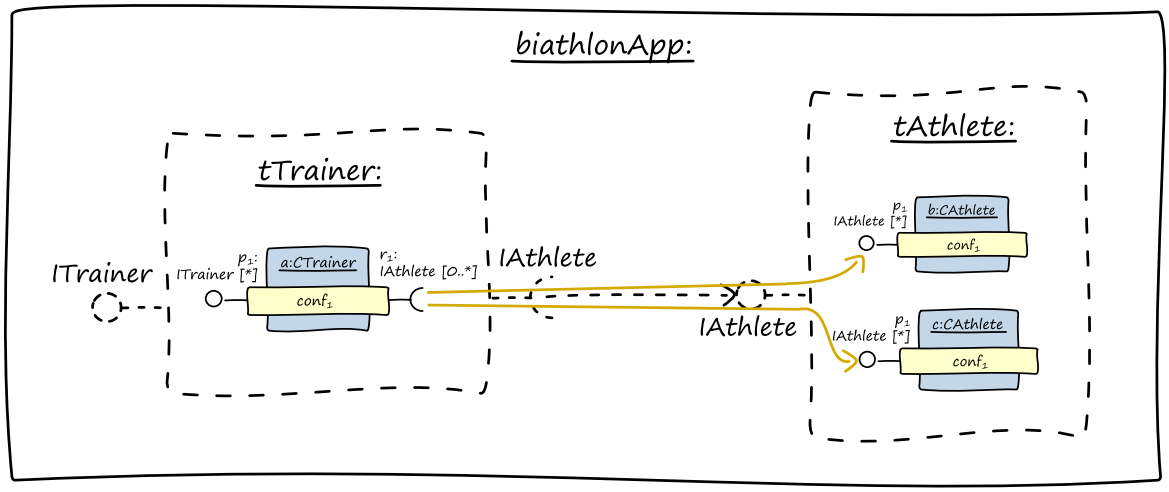


Abbildung 6-7: Erzeugung einer gültigen Konfiguration

Welche Anforderungen wie spezifiziert werden können und wie diese automatisiert durch das Framework umgesetzt werden, wird nun in den folgenden Abschnitten detailliert vorgestellt. Dann wird u.a. auch erläutert, wie beispielsweise innerhalb einer Anwendungsspezifikation *Template*-übergreifende Anforderungen hinterlegt werden können. So kann spezifiziert werden, dass die Anzahl von Sportlerkomponenten innerhalb einer Anwendung mit der Zahl von Schießbahnkomponenten übereinstimmen muss.

Im folgenden Abschnitt wird nun eine Erweiterung des Komponentenmodells vorgestellt, welche die Realisierung der zuvor vorgestellten Anforderungen ermöglicht. Anschließend wird gezeigt, wie das Framework basierend auf dem erweiterten Modell die spezifizierten Anforderungen automatisiert berücksichtigt.

6.2 Syntax einer Anwendungsspezifikation

Ein Ziel dieser Arbeit ist es, die Entwicklung von Anwendungen zu ermöglichen, die zum einen auf selbstkonfigurierenden Komponenten beruhen, und zum anderen Anwendungsarchitektur-spezifische Anforderungen während des Konfigurationsprozesses automatisiert umsetzen. In diesem Abschnitt wird hierzu eine Erweiterung des bereits aus Kapitel 4 und 5 bekannten Komponentenmodells vorgestellt, welches die Spezifikation derartiger Anforderungen

ermöglicht. Hierzu werden diejenigen Elemente des Komponentenmodells detailliert erläutert, welche der Spezifikation der zuvor beschriebenen Anforderungen dienen. Hierbei wird insbesondere die Syntax der einzelnen Spezifikationselemente vorgestellt, während die Semantik informell angegeben wird. Die formale Semantik der einzelnen Spezifikationselemente wird im nachfolgenden Abschnitt erläutert. Das der Lösung zugrundeliegende Komponentenmodell ist in Abbildung 6-8 dargestellt. Es handelt sich hierbei wiederum um eine Erweiterung des bereits aus den Kapiteln 4 und 5 bekannten Komponentenmodells.

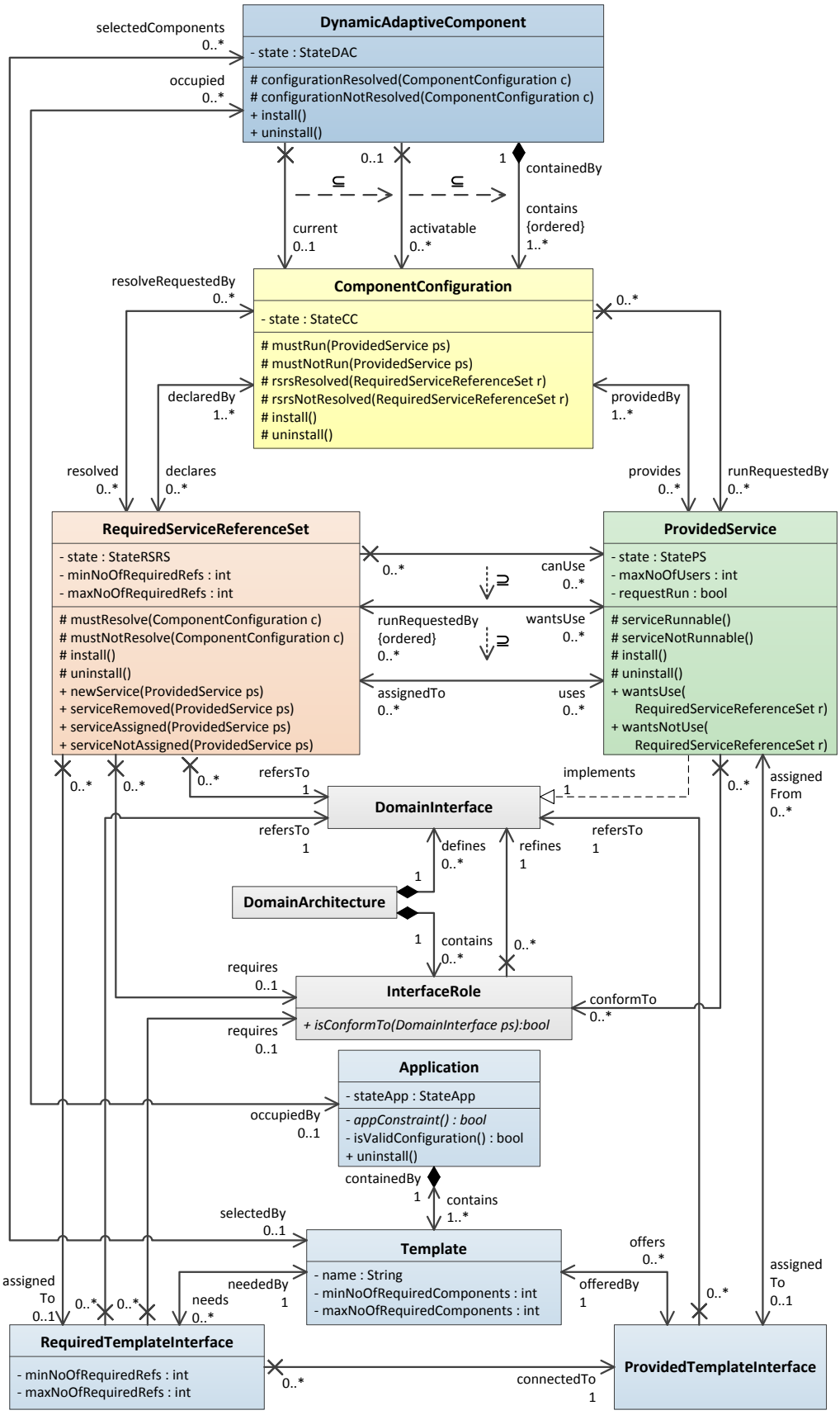


Abbildung 6-8: Das vollständige Komponentenmodell zur Spezifikation und Umsetzung anwendungsglobaler Eigenschaften

Das Modell wurde im Vergleich zum Modell aus Kapitel 5 um vier Klassen erweitert, nämlich um *Application*, *Template*, *RequiredTemplateInterface* und *ProvidedTemplateInterface*. Mit Hilfe dieser Klassen können die zuvor erläuterten Anforderungen spezifiziert werden. Die Klassen werden nun in den folgenden Abschnitten detailliert vorgestellt.

6.2.1 Die Klasse *Application*

Jede Instanz der Klasse *Application* repräsentiert eine Anwendungsinstanz. Sie definiert sämtliche Anforderungen an das zu erzeugende Komponentennetzwerk und steuert dessen Umsetzung. Innerhalb einer Anwendung wird zwischen den Zuständen RUNNING und NOT_RUNNING unterschieden. Der aktuelle Zustand einer Anwendung ist im Attribut *stateApp* der Klasse *Application* hinterlegt (siehe Formel 6-1):

$$\boldsymbol{StateApp} = \{NOT\_RUNNING, RUNNING\}$$
$$\boldsymbol{stateApp} \in StateApp$$

Formel 6-1

Eine Anwendung befindet sich immer dann im Zustand RUNNING, falls eine Konfiguration erzeugt werden konnte, welche alle Anwendungsarchitektur-spezifischen Anforderungen der Anwendung erfüllt. Ansonsten befindet sich die Anwendung im Zustand NOT_RUNNING.

Ein wesentlicher Schritt bei der Erzeugung von Anwendungskonfigurationen ist der, aus den zu Verfügung stehenden Komponenten diejenigen auszuwählen, die für die Anwendung in der aktuellen Situation benötigt werden. Hierbei spielen die bereits erwähnten *Templates* eine entscheidende Rolle. Mit Hilfe von *Templates* können Anwendungsentwickler Voraussetzungen definieren, unter denen Komponenten in eine Anwendung aufgenommen werden sollen.

Für eine Anwendung können beliebig viele solcher *Templates* definiert werden. Die Menge der *Templates* einer Anwendung ist im Modell aus Abbildung 6-8 durch die Assoziation *contains* repräsentiert (siehe Formel 6-2):

$$\boldsymbol{contains}: Application \rightarrow \mathcal{P}(Template)$$

Formel 6-2

So könnten für eine Biathlonanwendung beispielsweise folgende *Templates* definiert werden:

```
contains(biathlonApp) = { tTrainer, tAthlete, tLeftStick,
                          tRightStick, tShootingSupervisor,
                          tShootingLine }
```

Mit Hilfe des Prädikats *appConstraint* der Klasse *Application* lassen sich Anforderungen definieren, die über die Möglichkeiten der Anforderungsspezifikation durch *Templates* hinausgehen. Insbesondere können Anforderungen spezifiziert werden, die Komponenten mehrerer *Templates* gleichzeitig betreffen. Wie später gezeigt wird, kann mit Hilfe dieses Prädikats beispielsweise definiert werden, dass eine gültige Anwendungskonfiguration stets genauso viele Schießbahnkomponenten wie Sportlerkomponenten beinhalten muss. Eine solche Spezifikation könnte wie folgt aussehen (siehe Listing 6-1):

```
1  context Application :: appConstraint() : bool
2  post: result =
3    contains->select(name='tAthlete')->selectedComponents->size() ==
4    contains->select(name='tShootingLine')->selectedComponents->size()
```

Listing 6-1: Spezifikation des Prädikats *appConstraint* für eine Biathlonanwendung

Wie später noch erläutert wird, handelt es sich bei der Menge *selectedComponents* um diejenigen Komponenten, welche innerhalb eines *Templates* für die Anwendung verwendet werden. Mit Hilfe des Prädikats *appConstraint* kann ein Anwendungsentwickler Anforderungen an diese Menge definieren. Wie bereits angedeutet, handelt es sich bei *Templates* um das zentrale Spezifikationsmittel des hier vorgestellten Ansatzes. Die entsprechende Klasse des Komponentenmodells wird im folgenden Abschnitt vorgestellt.

6.2.2 Die Klasse *Template*

Mit Hilfe von *Templates* können Kriterien zur Auswahl von Komponenten für eine Anwendung spezifiziert werden. Im Modell aus Abbildung 6-8 wird jedes *Template* einer Anwendungsinstanz als Instanz der Klasse *Template* repräsentiert, wobei jedes *Template* genau einer Anwendung zugeordnet ist (siehe Formel 6-3):

$$\boldsymbol{containedBy}: Template \rightarrow Application$$

Formel 6-3

Wie später im Kapitel noch erläutert wird, ordnet das Framework zur Laufzeit jedem *Template* diejenigen Komponenten zu, welche die Anforderungen des jeweiligen *Templates* erfüllen. Diese sind anschließend mögliche Kandidaten für die Verwendung innerhalb der Anwendung. Hierbei kann es vorkommen, dass für eine Anwendung nur eine Teilmenge der Komponenten, die zu einem *Template* kompatibel sind, auch tatsächlich benötigt wird. So kann beispielsweise für die Biathlonanwendung die Anforderung bestehen, dass genau eine Schießaufsichtkomponente in die Anwendung eingebunden werden soll, obwohl zwei zur Verfügung stünden. Hierzu muss dem Anwendungsentwickler die Möglichkeit gegeben werden, die Anzahl der benötigten Komponenten für ein *Template* spezifizieren zu können. Zu diesem Zweck definiert die Klasse *Template* zwei Attribute, nämlich *minNoOfRequiredComponents* und *maxNoOfRequiredComponents* (siehe Formel 6-4):

$$\boldsymbol{minNoOfRequiredComponents}: Template \rightarrow \mathbb{N}_0$$
$$\boldsymbol{maxNoOfRequiredComponents}: Template \rightarrow \mathbb{N}_0 \cup \infty$$
$$\forall t \in Template:$$
$$minNoOfRequiredComponents(t) \leq maxNoOfRequiredComponents(t)$$

Formel 6-4

Entsprechend könnte die Spezifikation eines *Templates* für Schießaufsichtkomponenten wie folgt aussehen:

$$minNoOfRequiredComponents(tShootingSupervisor) = 1$$
$$maxNoOfRequiredComponents(tShootingSupervisor) = 1$$

Das Framework stellt während des Konfigurationsprozesses sicher, dass zur Laufzeit der Anwendung stets mindestens die im Attribut *minNoOfRequiredComponents* angegebene Anzahl an Komponenten verwendet wird. Außerdem wird sichergestellt, dass maximal so viele Komponenten eines *Templates* innerhalb der Anwendung verwendet werden, wie im Attribut *maxNoOfRequiredComponents* angegeben ist. Steht die benötigte Anzahl an Komponenten für ein *Template* der Anwendung nicht zur Verfügung, so wird die Anwendung nicht ausgeführt. Stehen mehr Komponenten für ein *Template* zur Verfügung, als im Attribut *maxNoOfRequiredComponents* angegeben, so verwendet das Framework eine Teilmenge der zur Verfügung stehenden Komponenten.

Jedes *Template* besitzt schließlich noch einen eindeutigen Namen, welcher im Attribut *name* hinterlegt ist (siehe Formel 6-5):

$$\boldsymbol{name}: Template \rightarrow String$$

Formel 6-5

Bei diesen Namen handelt es sich um die eindeutigen Bezeichner der Elemente der Menge *contains* (siehe Formel 6-2). Diese werden im Attribut *name* nochmals explizit hinterlegt, um sie, wie später gezeigt wird, zur Spezifikation *Template*-übergreifender Anforderungen adressieren zu können. Hierauf wird insbesondere bei der Spezifikation des Prädikats *appConstraint* zurückgegriffen, welches bereits im vorherigen Abschnitt eingeführt wurde.

Wie bereits erwähnt, definiert jedes *Template* Kriterien, unter denen eine Komponente einem *Template* zugeordnet werden kann. Innerhalb des Modells werden zwei Arten von Anforderungen unterschieden, nämlich zum einen Anforderungen an die Struktur von Komponenten, und zum anderen Anforderungen an die Rückgabewerte von Methoden, die über angebotene Dienste von Komponenten implementiert werden. Anforderungen an die Komponentenstruktur beziehen sich auf die Existenz von *ProvidedServices* und *RequiredServiceReferenceSets* von Komponenten. Diese Anforderungen werden über die Klassen *RequiredTemplateInterface* und *ProvidedTemplateInterface* des Modells aus Abbildung 6-8 spezifiziert. Beide Klassen werden in den folgenden zwei Abschnitten noch vorgestellt.

Anforderungen, welche die angebotenen Dienste von Komponenten betreffen, werden mit Hilfe der Klasse *ProvidedTemplateInterface* spezifiziert. Jede Instanz dieser Klasse repräsentiert die Anforderungen an genau einen angebotenen Dienst von Komponenten. Wie später noch erläutert wird, kann mit dessen Hilfe beispielsweise für ein Trainer-*Template* gefordert werden, dass ihm nur solche Komponenten zugeordnet werden sollen, die einen Dienst anbieten, welcher die Domänenschnittstelle *ITrainer* implementiert. Jedes *Template* kann beliebig viele Instanzen der Klasse *ProvidedTemplateInterface* definieren. Diese sind im Modell aus Abbildung 6-8 durch die Assoziation *offers* repräsentiert (siehe Formel 6-6):

$$\boldsymbol{offers}: Template \rightarrow \mathcal{P}(ProvidedTemplateInterface)$$

Formel 6-6

Mit Hilfe der Klasse *RequiredTemplateInterface* können Anforderungen bzgl. Abhängigkeiten zu Diensten anderer Komponenten spezifiziert werden. So kann beispielsweise für ein Trainer-*Template* spezifiziert werden, dass jede ihr zugeordnete Komponente für die Anwendung mit mindestens einem Athletendienst verbunden sein muss, der wiederum durch eine Komponente angeboten wird, die einem Sportler-*Template* zugeordnet ist (mehr hierzu

später). Jede Instanz der Klasse *RequiredTemplateInterface* definiert die Anforderungen genau einer solchen Abhängigkeit. Ein *Template* darf beliebig viele dieser *RequiredTemplateInterfaces* definieren. Diese werden im Modell durch die Assoziation *needs* repräsentiert (siehe Formel 6-7):

$$\boldsymbol{needs}: Template \rightarrow \mathcal{P}(RequiredTemplateInterface)$$

Formel 6-7

Abschließend werden in den zwei folgenden Abschnitten noch die Klassen *ProvidedTemplateInterface* und *RequiredTemplateInterface* erläutert.

6.2.3 Die Klasse *ProvidedTemplateInterface*

Mit Hilfe der Klasse *ProvidedTemplateInterface* des Modells lassen sich Anforderungen an Komponenten formulieren, die sich auf angebotene Dienste dieser Komponenten beziehen. So kann mit dessen Hilfe beispielsweise ein *Template* definiert werden, welches nur solche Komponenten zulässt, die einen Dienst anbieten, der die Domänenschnittstelle *IStick* implementiert. Jede Instanz der Klasse *ProvidedTemplateInterface* repräsentiert genau eine solche Anforderung, wobei für ein *Template* beliebig viele *ProvidedTemplateInterfaces* spezifiziert werden dürfen. Diese sind im Modell über die Assoziation *offeredBy* verbunden (siehe Formel 6-8):

$$\boldsymbol{offeredBy}: ProvidedTemplateInterface \rightarrow Template$$

Formel 6-8

Des Weiteren referenziert jedes *ProvidedTemplateInterface* genau eine Domänenschnittstelle über die Assoziation *refersTo* (siehe Formel 6-9):

$$\boldsymbol{refersTo}: ProvidedTemplateInterface \rightarrow DomainInterface$$

Formel 6-9

Die Spezifikation eines *Templates* für Skistockkomponenten könnte wie folgt aussehen:

```
Template = { tLeftStick }
ProvidedTemplateInterface = {pStick}
offers(tLeftStick) = {pStick}
offeredBy(pStick) = tLeftStick
refersTo(pStick) = IStick
```

Damit eine Komponente diesem *Template* durch das Framework zugeordnet wird, muss sie somit einen Dienst anbieten, der die Domänenschnittstelle *IStick* implementiert. Die exakte Semantik von *ProvidedTemplateInterface* wird im weiteren Verlauf dieses Kapitels noch näher beschrieben.

6.2.4 Die Klasse *RequiredTemplateInterface*

Mit Hilfe von *RequiredTemplateInterfaces* lassen sich Anforderungen an *RequiredServiceReferenceSets* von Komponenten stellen. Insbesondere lässt sich mit deren Hilfe das Intervall, welches in *RequiredServiceReferenceSets* von Komponenten durch die Attribute *minNoOfRequiredRefs* und *maxNoOfRequiredRefs* definiert ist, je nach Bedarf der Anwendung

einschränken. Hierzu definiert die Klasse *RequiredTemplateInterface* zwei Attribute, nämlich *minNoOfRequiredRefs* und *maxNoOfRequiredRefs* (siehe Formel 6-10):

$$\boldsymbol{minNoOfRequiredRefs}: RequiredTemplateInterface \rightarrow \mathbb{N}_0$$
$$\boldsymbol{maxNoOfRequiredRefs}: RequiredTemplateInterface \rightarrow \mathbb{N}_0 \cup \infty$$
$$\forall rti \in RequiredTemplateInterface:$$
$$minNoOfRequiredRefs(rti) \leq maxNoOfRequiredRefs(rti)$$

Formel 6-10

Jedes *RequiredTemplateInterface* ist genau einem *Template* über die Assoziation *neededBy* zugeordnet (siehe Formel 6-11):

$$\boldsymbol{neededBy}: RequiredTemplateInterface \rightarrow Template$$
$$\forall rti \in RequiredTemplateInterface: (neededBy(rti) = t) \Longrightarrow rti \in needs(t)$$

Formel 6-11

Des Weiteren referenziert jedes *RequiredTemplateInterface* genau eine Domänenschnittstelle. Die ist im Modell durch die Assoziation *refersTo* repräsentiert. Diese Referenz dient später dem Framework dazu, jedes *RequiredServiceReferenceSet* einem passenden *RequiredTemplateInterface* zuzuordnen. Zusätzlich kann ein *RequiredTemplateInterface* noch optional eine *InterfaceRole* referenzieren. Dieser Sachverhalt ist im Modell über die Assoziation *requires* repräsentiert. Die beiden Assoziationen *refersTo* und *requires* sind formalisiert in Formel 6-12 angegeben:

$$\boldsymbol{refersTo}: RequiredTemplateInterface \rightarrow DomainInterface$$
$$\boldsymbol{requires}: RequiredTemplateInterface \rightarrow InterfaceRole$$

Formel 6-12

Jedes *RequiredTemplateInterface* definiert zudem, welchem *Template* diejenige Komponente zugeordnet sein muss, dessen Dienst durch ein *RequiredServiceReferenceSet* verwendet wird. Hierzu ist jedes *RequiredTemplateInterface* mit genau einem *ProvidedTemplateInterface* derselben Anwendung verbunden. Diese Verbindung ist im Modell durch die Assoziation *connectedTo* repräsentiert (siehe Formel 6-13):

$$\boldsymbol{connectedTo}: RequiredTemplateInterface \rightarrow ProvidedTemplateInterface$$
$$\forall rti \in RequiredTemplateInterface: (connectedTo(rti) = pti)$$
$$\Longrightarrow containedBy\big(neededBy(rti)\big) = containedBy\big(offeredBy(pti)\big)$$

Formel 6-13

Zusätzlich ist zu beachten, dass ein *RequiredTemplateInterface* nur mit einem *ProvidedTemplateInterface* über *connectedTo* verbunden werden darf, welches dieselbe Domänenschnittstelle referenziert (siehe Formel 6-14):

$$\forall rti \in RequiredTemplateInterface: refersTo(rti) = refersTo\big(connectedTo(rti)\big)$$

Formel 6-14

Mit Hilfe der vorgestellten Mittel können nun für eine Anwendung beispielsweise zwei *Templates* definiert werden, eines für Trainerkomponenten und eines für Sportlerkomponenten:

Template = {tTrainer, tAthlete}
ProvidedTemplateInterface = {pTrainer, pAthlete}
RequiredTemplateInterface = {rAthlete}
offers(tTrainer) = {pTrainer}, offeredBy(pTrainer) = tTrainer
refersTo(pTrainer) = ITrainer
needs(tTrainer) = {rAthlete}, neededBy(rAthlete) = tTrainer
refersTo(rAthlete) = IAthlete
offers(tAthlete) = {pAthlete}, offeredBy(pAthlete) = tAthlete
refersTo(pAthlete) = IAthlete
connectedTo(rAthlete) = pAthlete

In diesem Fall ist das Trainer-*Template* über das *RequiredTemplateInterface rAthlete* mit dem Athleten-*Template* verknüpft. Wie später noch gezeigt wird, wird das Framework die Komponenten innerhalb des *Templates tTrainer* mit denjenigen des *Templates tAthlete* verbinden. Die genaue Semantik dieser Assoziation wird im weiteren Verlauf des Kapitels allerdings noch genauer erläutert. Zunächst wird nun im folgenden Abschnitt noch die grafische Repräsentation der einzelnen Spezifikationselemente anhand eines Beispiels vorgestellt.

6.2.5 Grafische Darstellung der Spezifikationselemente

Um Anwendungsspezifikationen übersichtlich und kompakt darstellen zu können, wird im weiteren Verlauf der Arbeit eine einheitliche Notation für die zuvor vorgestellten Spezifikationselemente definiert. Abbildung 6-9 zeigt ein Beispiel für eine mögliche Anwendungsspezifikation einer Biathlonanwendung.

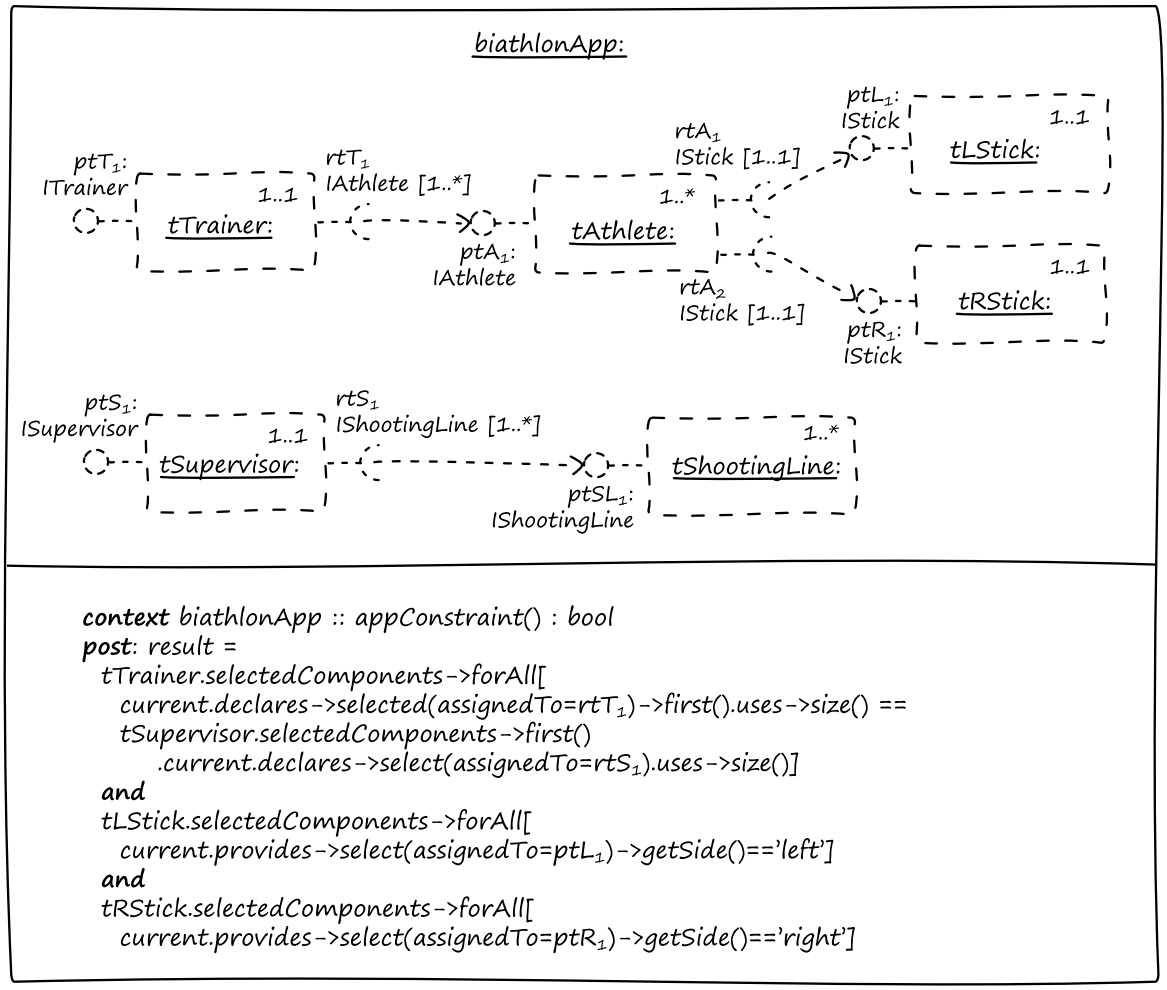


Abbildung 6-9: Beispiel der grafischen Darstellung einer Anwendungsspezifikation

Eine Anwendung wird als Rechteck dargestellt, wobei der Name der Anwendung oben notiert wird. Jedes *Template* wird als gestricheltes Rechteck notiert, welches den Namen des *Templates* beinhaltet. Innerhalb eines *Templates* werden die Inhalte der Attribute *minNoOfRequiredComponents* und *maxNoOfRequiredComponents* oben rechts angegeben. Ein *ProvidedTemplateInterface* wird als gestrichelter Kreis notiert, welcher mit dem Namen sowie der referenzierten Domänenschnittstelle beschriftet ist. *RequiredTemplateInterfaces* werden entsprechend als gestrichelte Halbkreise dargestellt. Diese werden ebenfalls mit der referenzierten Domänenschnittstelle, ggf. der referenzierten Schnittstellenrolle sowie mit dem Namen beschriftet. Verbindungen zwischen einem *RequiredTemplateInterface* und einem *ProvidedTemplateInterface* (*connectedTo*) werden mit Hilfe eines gestrichelten Pfeils visualisiert. Die Spezifikation des Prädikats *appConstraint* wird in einem gesonderten Bereich unterhalb der *Templates* angegeben. Die Darstellung wird im weiteren Verlauf des Kapitels an geeigneter Stelle noch um weitere Notationselemente erweitert (z.B. zur Visualisierung der Zugehörigkeit von Komponenten zu *Templates*). Im folgenden Abschnitt wird nun die Semantik der einzelnen Spezifikationselemente im Detail vorgestellt.

6.3 Semantik einer Anwendungsspezifikation

Während im vorherigen Abschnitt im Wesentlichen die Elemente der Spezifikationssprache vorgestellt und die Semantik ausschließlich informell angegeben wurde, wird in diesem

Abschnitt letztere behandelt. Hierbei wird insbesondere dargestellt, in welcher Form die einzelnen Spezifikationselemente Einfluss auf die Anwendungskonfiguration nehmen.

Ziel des Frameworks ist es, eine Anwendungskonfiguration zu erzeugen, welche alle spezifizierten Anforderungen erfüllt. Sobald dies gelungen ist, wird die Anwendung vom Zustand NOT_RUNNING in den Zustand RUNNING versetzt. Oder anders ausgedrückt: Befindet sich eine Anwendung im Zustand RUNNING, so erfüllt die erzeugte Anwendungskonfiguration alle definierten Anwendungsarchitektur-spezifischen Anforderungen. Die Semantik der einzelnen Spezifikationselemente wird basierend auf den Eigenschaften definiert, die eine Anwendung im Zustand RUNNING besitzt.

Die Bedingung, unter denen eine Anwendungskonfiguration sämtliche Anforderungen erfüllt, die durch die Spezifikation gegeben sind, wird im Folgenden durch das Prädikat *isValidConfiguration* beschrieben (siehe Formel 6-15):

$$isValidConfiguration: Application \rightarrow bool$$

Formel 6-15

Wie bereits zuvor erläutert wurde, spielen *Templates* im vorgestellten Konzept eine zentrale Rolle bei der Spezifikation von Anforderungen an Anwendungskonfigurationen. Durch sie werden Kriterien zur Auswahl geeigneter Komponenten für eine Anwendung spezifiziert, sowie Anforderungen an die Verschaltung der Komponenten definiert.

Das Framework wird später versuchen eine Konfiguration zu erzeugen, welche die Anforderungen eines jeden *Templates* erfüllt, indem ihnen geeignete Komponenten zugeordnet werden und entsprechende Verbindungen zu anderen Komponenten hergestellt werden. Zur Spezifikation der Bedingungen, unter denen die Anforderungen eines *Templates* als erfüllt angesehen werden können, wird zunächst ein Hilfsprädikat namens *isResolved* definiert, welches bei gegebenem *Template* dann *wahr* zurückliefert, falls all seine Anforderungen erfüllt sind und sonst *falsch* (siehe Formel 6-16):

$$\boldsymbol{isResolved}: Template \rightarrow bool$$

Formel 6-16

Eine Konfiguration wird dann als konform zur Architekturspezifikation bezeichnet, wenn zum einen die Anforderungen aller *Templates* der Anwendung erfüllt werden, und zum anderen das Prädikat *appConstraint* zu *wahr* ausgewertet wird (siehe Formel 6-17):

$$\forall app \in Application:$$
$$isValidConfiguration(app) \Leftrightarrow \begin{bmatrix} \big(\forall t \in contains(app): isResolved(t)\big) \wedge \\ appConstraint(\,) \end{bmatrix}$$

Formel 6-17

Entscheidend für die Bedingungen, unter denen eine Anwendungskonfiguration bei gegebener Anwendungsspezifikation gültig ist, ist somit neben dem Prädikat *appConstraint* noch das Prädikat *isResolved*. Im weiteren Verlauf des Abschnitts wird nun gezeigt, unter welchen Bedingungen dieses Prädikat zu *wahr* ausgewertet wird.

Zur Erzeugung einer gültigen Anwendungskonfiguration muss sie die Anforderungen aller *Templates* der Anwendungen erfüllen. Hierzu müssen für jedes *Template* Komponenten gefunden werden, die die jeweils definierten Anforderungen erfüllen. Im Modell aus Abbildung 6-8 ist die Menge derjenigen Komponenten, welche für ein *Template* zur Verwendung innerhalb der Anwendung durch das Framework ausgewählt wurden, durch die Assoziation s*elected*Components repräsentiert (siehe Formel 6-18):

$$\boldsymbol{selectedComponents}: Template \rightarrow \mathcal{P}(DynamicAdaptiveComponent)$$
$$\boldsymbol{selectedBy}: DynamicAdaptiveComponent \rightarrow Template$$

Formel 6-18

Im weiteren Verlauf werden nun die Bedingungen vorgestellt, die Elemente dieser Menge sowie die Menge selber erfüllen muss, damit die Anforderungen eines *Templates* als erfüllt gelten können.

Eine der wesentlichen Bedingungen, die an die Menge *selectedComponents* formuliert werden können bezieht sich auf die geforderte Anzahl an Elementen der Menge. Diese wird durch die bereits vorgestellten Attribute *minNoOfRequiredComponents* und *maxNoOfRequiredComponents* spezifiziert. Die erste Bedingung, unter der das Prädikat *isResolved* zu *wahr* ausgewertet wird ist in Formel 6-19 angegeben:

$$\forall t \in Template: isResolved(t) \Longrightarrow$$
$$minNoOfRequiredComponents(t) \leq |selectedComponents(t)| \leq maxNoOfRequiredComponents(t)$$

Formel 6-19

Mit den Attributen *minNoOfRequiredComponents* und *maxNoOfRequiredComponents* kann ein Anwendungsentwickler somit spezifizieren, wie viele Komponenten für ein *Template* mindestens bzw. maximal benötigt werden.

Bei Elementen der Menge *selectedComponents* handelt es sich um diejenigen Komponenten, welche innerhalb einer Anwendungskonfiguration verwendet werden. Damit eine Komponente verwendet werden kann, muss sie ausgeführt werden. Hierzu muss sich eine Komponente im Zustand RESOLVED befinden (siehe Kapitel 4). Diese Eigenschaft kann wie folgt formal definiert werden (siehe Formel 6-20):

$$\forall t \in Template: isResolved(t) \Longrightarrow$$
$$\forall c \in selectedComponents(t): state(c) = RESOLVED$$

Formel 6-20

Befindet sich eine Komponente im Zustand RESOLVED, so besitzt sie eine aktive Konfiguration, welche im Modell durch die bereits vorgestellte Assoziation *current* repräsentiert ist. An diese Konfiguration stellt ein *Template* weitere Anforderungen.

Die erste Bedingung, die eine Komponente zur Aufnahme in die Menge *selectedComponents* eines *Templates* erfüllen muss, betrifft die Struktur der Komponente bzw. der aktiven *ComponentConfiguration*. Zusammenfassend ausgedrückt muss die aktive *ComponentConfiguration* bestimmte, durch ein *Template* vorgegebene *ProvidedServices* und *RequiredServiceReferenceSets* definieren. Diese Anforderungen werden mit Hilfe der bereits

vorgestellten Spezifikationselemente *ProvidedTemplateInterface* und *RequiredTemplateInterface* durch den Anwendungsentwickler angegeben. Zur Definition der Bedingungen, unter denen eine *ComponentConfiguration* die Anforderungen erfüllt, wird zunächst ein Prädikat *isTemplateCompliant* definiert (siehe Formel 6-21):

$$\boldsymbol{isTemplateCompliant}: Template \times ComponentConfiguration \rightarrow bool$$

Formel 6-21

Das Prädikat wird dann zu *wahr* ausgewertet, wenn die als Parameter übergebene *ComponentConfiguration* alle Anforderungen eines *Templates* erfüllt. Dieses Prädikat wird in Abschnitt 6.3.1 näher spezifiziert. Das Prädikat *isResolved* eines *Templates* wird nur dann zu *wahr* ausgewertet, falls für jedes Element der Menge *selectedComponents* das Prädikat *isTemplateCompliant* zu *wahr* ausgewertet wird (siehe Formel 6-22):

$$\forall t \in Template: isResolved(t) \Longrightarrow \\ \forall c \in selectedComponents(t): isTemplateCompliant\big(t, current(c)\big)$$

Formel 6-22

Neben der Auswahl geeigneter Komponenten für eine Anwendungskonfiguration spielt die anforderungskonforme Verschaltung dieser Komponenten eine wichtige Rolle. Anforderungen an die Verschaltung von Komponenten werden mit Hilfe der Klasse *RequiredTemplateInterface* sowie der Assoziation *connectedTo* spezifiziert. Für eine Komponente gelten die Abhängigkeiten zu Diensten anderer Komponenten nur dann als aufgelöst, wenn hierbei die Vorgaben des jeweiligen *Templates* bzw. des *RequiredTemplateInterfaces* berücksichtigt wurden. Hierbei muss jedes *RequiredServiceReferenceSet* der aktiven *ComponentConfiguration* die Bedingungen des *Templates* erfüllen. Diese werden im Folgenden mit Hilfe des Prädikats *isDependencyResolved* definiert (siehe Formel 6-23):

$$\boldsymbol{isDependencyResolved}: Template \times RequiredServiceReferenceSet \rightarrow bool$$

Formel 6-23

Die Bedingungen, unter denen das Prädikat zu *wahr* ausgewertet wird, werden im Abschnitt 6.3.2 im Detail erläutert. Mit Hilfe dieses Prädikats lässt sich nun auch die letzte Bedingung formulieren, unter denen das Prädikat *isResolved* zu *wahr* ausgewertet werden kann (siehe Formel 6-24):

$$\forall t \in Template: isResolved(t) \Longrightarrow \\ \forall c \in selectedComponents(t), \forall rsrs \in declares\big(current(c)\big): \\ isDependencyResolved(t, rsrs)$$

Formel 6-24

Zusammenfassend kann das Prädikat *isResolved* nun wie in Formel 6-25 angegeben definiert werden:

$$\forall t \in Template: isResolved(t) \Leftrightarrow \\ \left[\begin{matrix} minNoOfRequiredComponents(t) \leq |selectedComponents(t)| \\ \leq maxNoOfRequiredComponents(t) \end{matrix}\right] \wedge \\ [\forall c \in selectedComponents(t): state(c) = RESOLVED] \wedge \\ \left[\forall c \in selectedComponents(t): isTemplateCompliant\big(t, current(c)\big)\right] \wedge \\ \left[\begin{matrix} \forall c \in selectedComponents(t), \forall rsrs \in declares\big(current(c)\big): \\ isDependencyResolved(t, rsrs) \end{matrix}\right]$$

Formel 6-25

Um die Semantik der vorgestellten Spezifikationssprache zu vervollständigen, müssen nun noch die Prädikate *isTemplateCompliant* und *isDependencyResolved* definiert werden. Dies wird nun in den zwei folgenden Abschnitten nachgeholt.

6.3.1 Das Prädikat *isTemplateCompliant*

Mit Hilfe des Prädikats *isTemplateCompliant* werden Bedingungen formuliert, die eine Komponente erfüllen muss, damit sie innerhalb eines *Templates* in eine Anwendung aufgenommen werden kann. Hierbei adressiert das Prädikat insbesondere Anforderungen, die die Struktur einer Komponente bzw. der aktiven Konfiguration betreffen.

Anforderungen an angebotene Dienste einer Komponente werden mit Hilfe der Klasse *ProvidedTemplateInterface* spezifiziert. Hierbei ist eine Komponente bzw. die aktive Konfiguration der Komponente nur dann kompatibel zu einem *Template*, wenn die *ComponentConfiguration* für jedes *ProvidedTemplateInterface* des *Templates* einen Dienst anbietet, welcher die referenzierte Domänenschnittstelle implementiert. Es muss also eine surjektive Abbildung von *ProvidedServices* zu *ProvidedTemplateInterfaces* existieren.

Die Klasse *ProvidedTemplateInterface* definiert neben Anforderungen an die Existenz bestimmter angebotener Dienste einer *ComponentConfiguration* zusätzlich Anforderungen an den Ausführungszustand der betroffenen Dienste. So wird das Prädikat *isTemplateCompliant* nur dann zu *wahr* ausgewertet, wenn sich jeder *ProvidedService* der betreffenden *ComponentConfiguration*, zu der ein entsprechendes *ProvidedTemplateInterface* existiert, im Zustand RUNNING befindet. Auf diese Weise kann ein Anwendungsentwickler mit Hilfe von *ProvidedTemplateInterfaces* sicherstellen, dass bestimmte Dienste einer Komponente ausgeführt werden, obwohl das Flag *requestRun* des jeweiligen Dienstes nicht gesetzt ist. Die entsprechenden Bedingungen sind in Formel 6-26 zusammengefasst:

$$\forall t \in Template, \forall config \in ComponentConfiguration: \\ isTemplateCompliant(t, config) \Longrightarrow \\ \left[\begin{matrix} \forall pti \in offers(t) \exists ps \in provides(config): \\ refersTo(pti) = implements(ps) \wedge \\ state(ps) = RUNNING \end{matrix}\right]$$

Formel 6-26

Wie soeben bereits erwähnt, muss eine surjektive Abbildung von *ProvidedServices* zu *ProvidedTemplateInterfaces* existieren, so dass die Bedingungen in Formel 6-26 erfüllt sind. Hierbei darf ein *ProvidedService* maximal einem *ProvidedTemplateInterface* zugeordnet sein. Diese Zuordnung ist im Modell aus Abbildung 6-8 durch die Assoziation *assignedTo* repräsentiert (siehe Formel 6-27):

$$\boldsymbol{assignedTo}: ProvidedService \rightarrow ProvidedTemplateInterface$$

Formel 6-27

Diese Funktion liefert für ein *ProvidedService* entweder die leere Menge zurück, oder das zugeordnete *ProvidedTemplateInterface.* Zur Verdeutlichung der soeben beschriebenen Sachverhalte sei ein *Template* angenommen, welches ein *ProvidedTemplateInterface* definiert, das die Domänenschnittstelle *IStick* referenziert (siehe Abbildung 6-10).

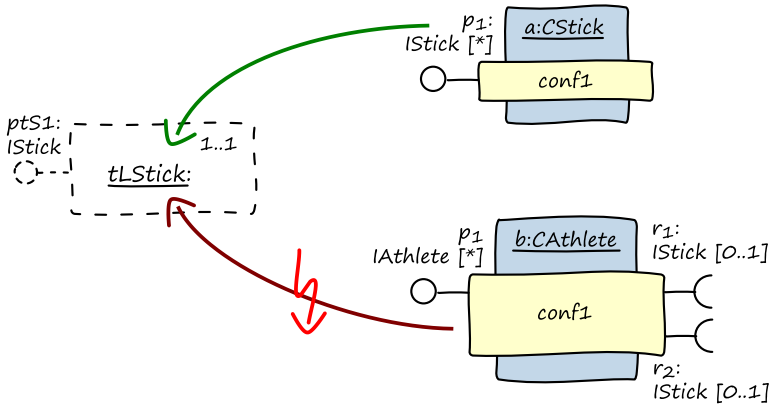


Abbildung 6-10: Beispiel für Anforderungen an angebotene Dienste von Komponenten

Die Komponente *a* bzw. dessen *ComponentConfiguration* erfüllt die Bedingung aus Formel 6-26, da sie einen Dienst anbietet, der die Schnittstelle *IStick* implementiert. Die Funktion *assignedTo* würde in diesem Fall *ptS1* zurückliefern. Die Komponente *b* hingegen bietet keinen entsprechenden Dienst an und kommt somit für eine Verwendung innerhalb des *Templates tLStick* nicht in Frage.

Neben Anforderungen an angebotene Dienste von Komponenten können auch Bedingungen an *RequiredServiceReferenceSets* von Komponenten bzw. deren aktiver *ComponentConfiguration* definiert werden. Derartige Anforderungen werden mit Hilfe der Klasse *RequiredTemplateInterface* spezifiziert. So wird das Prädikat *isTemplateCompliant* nur dann zu *wahr* ausgewertet, wenn zu jedem *RequiredTemplateInterface* ein *RequiredServiceReferenceSet* durch die *ComponentConfiguration* definiert wird, welches dieselbe Domänenschnittstelle referenziert. Zusätzlich muss zu jedem *RequiredServiceReferenceSet* ein *RequiredTemplateInterface* existieren, welches dieselbe Domänenschnittstelle referenziert. Es muss somit eine surjektive Abbildung zwischen *RequiredTemplateInterfaces* und *RequiredServiceReferenceSets* existieren. Ein *RequiredTemplateInterface* kann außerdem nicht nur eine Domänenschnittstelle referenzieren, sondern ergänzend hierzu auch eine Schnittstellenrolle (siehe Kapitel 5). Und hier gilt dasselbe, was auch für die Referenz auf Domänenschnittstellen gilt. Diese Bedingungen sind in Formel 6-28 zusammenfassend angegeben:

$$\forall t \in Template, \forall config \in ComponentConfiguration:$$
$$isTemplateCompliant(t, config) \Longrightarrow$$
$$\left[\forall rti \in needs(t) \exists rsrs \in declares(config): \begin{pmatrix} refersTo(rti) = refersTo(rsrs) \wedge \\ requires(rti) = requires(rsrs) \end{pmatrix}\right] \wedge$$
$$\left[\forall rsrs \in declares(config) \exists rti \in needs(t): \begin{pmatrix} refersTo(rti) = refersTo(rsrs) \wedge \\ requires(rti) = requires(rsrs) \end{pmatrix}\right]$$

Formel 6-28

Auf diese Weise kann ein Anwendungsentwickler Einfluss darauf nehmen, welche Abhängigkeiten eine Komponenten bzw. dessen aktive *ComponentConfiguration* zu Diensten anderer Komponenten besitzen darf bzw. muss.

In Abbildung 6-11 ist zur Verdeutlichung der Bedingung ein *Template* für Athletenkomponenten angegeben.

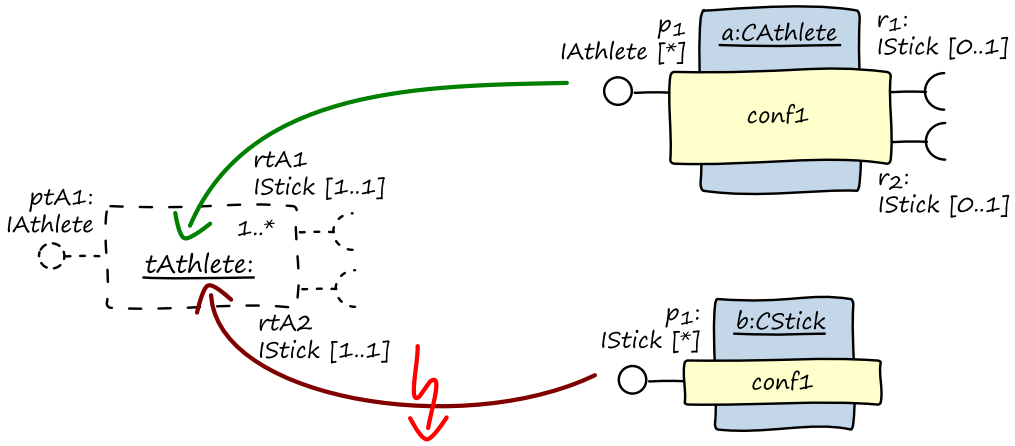


Abbildung 6-11: Beispiel für Anforderungen an *RequiredServiceReferenceSets* von Komponenten

Die *ComponentConfiguration* der Komponente *a* erfüllt hierbei alle Bedingungen, da sie sowohl die erforderliche Athletenschnittstelle anbietet, als auch die geforderten Skistockschnittstellen benötigt. Auch die Intervalle an den *RequiredServiceReferenceSets* erfüllen die Anforderungen des *Templates*, da sich die Intervalle überschneiden (siehe nächster Abschnitt). Die Komponentenkonfiguration der Komponente *b* erfüllt hingegen keine der Anforderungen.

Das Prädikat *isTemplateCompliant* ist hiermit vollständig definiert und in Formel 6-29 zusammenfassend angegeben:

$$\forall t \in Template, \forall config \in ComponentConfiguration:$$
$$isTemplateCompliant(t, config) \Longleftrightarrow$$
$$\left[\begin{array}{c} \forall pti \in offers(t) \exists ps \in provides(config): \\ refersTo(pti) = implements(ps) \wedge \\ state(ps) = RUNNING \end{array}\right] \wedge$$
$$\left[\forall rti \in needs(t) \exists rsrs \in declares(config): \begin{pmatrix} refersTo(rti) = refersTo(rsrs) \wedge \\ requires(rti) = requires(rsrs) \end{pmatrix}\right] \wedge$$
$$\left[\forall rsrs \in declares(config) \exists rti \in needs(t): \begin{pmatrix} refersTo(rti) = refersTo(rsrs) \wedge \\ requires(rti) = requires(rsrs) \end{pmatrix}\right]$$

Formel 6-29

Wie im nächsten Abschnitt gezeigt wird, können mit Hilfe der Klasse *RequiredTemplateInterface* zusätzlich Anforderungen an Verbindungen zwischen Komponenten spezifiziert werden. Hierzu wurde bereits das Prädikat *isDependencyResolved* vorgestellt (siehe Formel 6-23), jedoch nicht definiert. Dies wird nun im folgenden Abschnitt nachgeholt.

6.3.2 Das Prädikat *isDependencyResolved*

Mit Hilfe des Prädikats *isDependencyResolved* (siehe Formel 6-23) werden Bedingungen an *RequiredServiceReferenceSets* von Komponenten definiert, welche die Eigenschaften von Diensten der Menge *uses* betreffen. Jedes *RequiredServiceReferenceSet* einer aktiven *ComponentConfiguration* verwendet eine Menge von Diensten, welche von anderen Komponenten angeboten werden. Diese Menge ist im Modell über die bereits vorgestellte Menge *uses* repräsentiert. Mit Hilfe von *RequiredTemplateInterfaces* kann der Anwendungsentwickler nun bestimmte Anwendungsarchitektur-spezifische Anforderungen an die Elemente der Menge *uses* definieren. Diese werden nun im weiteren Verlauf dieses Abschnitts vorgestellt.

Jedes *RequiredTemplateInterface* definiert Anforderungen für genau ein *RequiredServiceReferenceSet* jeder zugeordneten Komponente. Zu jedem Zeitpunkt ist jedes *RequiredServiceReferenceSet* einer Komponente, welche einem *Template* zugeordnet ist, genau einem *RequiredTemplateInterface* zugeordnet und umgekehrt. Im Folgenden sei nun angenommen, dass die Funktion *assignedTo* zu einem *RequiredServiceReferenceSet* das zugeordnete *RequiredTemplateInterface* zurückliefert (siehe Formel 6-30). Hierbei handelt es sich also um eine bijektive Funktion.

$$\boldsymbol{assignedTo}: RequiredServiceReferenceSet \rightarrow RequiredTemplateInterface$$

Formel 6-30

Wie bereits im Abschnitt 6.2.4 vorgestellt wurde, definiert die Klasse *RequiredTemplateInterface* die Attribute *minNoOfRequiredRefs* und *maxNoOfRequiredRefs*. Mit Hilfe dieser Attribute kann der Anwendungsentwickler die Anzahl der zu verwendenden Dienste innerhalb der Menge *uses* eines *RequiredServiceReferenceSets* bestimmen. Jedes *RequiredServiceReferenceSet* muss mit mindestens so vielen Diensten anderer Komponenten verbunden sein, wie sich aus der Anzahl des Maximums des Attributs *minNoOfRequiredRefs* des *RequiredServiceReferenceSets* und des Attributes *minNoOfRequiredRefs* des zugeordneten *RequiredTemplateInterfaces* ergibt. Ebenso darf ein *RequiredServiceReferenceSet* maximal so viele Dienste anderer Komponenten nutzen, wie sich aus dem Minimum des Attributs *maxNoOfRequiredRefs* des *RequiredServiceReferenceSets* und des Attributs *maxNoOfRequiredRefs* des zugeordneten *RequiredTemplateInterfaces* ergibt. Diese Bedingungen sind in Formel 6-31 zusammengefasst:

$$\begin{aligned}
&\forall t \in Template, \forall rsrs \in RequiredServiceReferenceSet: \\
&\quad isDependendyResolved(t, rsrs) \Longrightarrow \\
&\qquad \max\begin{pmatrix} minNoOfRequiredRefs(rsrs), \\ minNoOfRequiredRefs\big(rsrsAssignedTo(rsrs)\big) \end{pmatrix} \leq \\
&\qquad\qquad |uses(rsrs)| \leq \\
&\qquad \min\begin{pmatrix} maxNoOfRequiredRefs(rsrs), \\ maxNoOfRequiredRefs\big(assignedTo(rsrs)\big) \end{pmatrix}
\end{aligned}$$

Formel 6-31

Mit Hilfe von *RequiredTemplateInterfaces* kann ein Anwendungsentwickler noch eine weitere Bedingung an *RequiredServiceReferenceSets* definieren. Diese bezieht sich darauf, mit welchen Diensten ein *RequiredServiceReferenceSet* verbunden werden darf.

Zum einen kann ein *RequiredTemplateInterface*, wie bereits zuvor erläutert, neben einer Domänenschnittstelle zusätzlich eine Schnittstellenrolle referenzieren. Ist dies der Fall, so darf ein dem *RequiredTemplateInterface* zugeordnetes *RequiredServiceReferenceSet* nur mit solchen Diensten verbunden sein, die diese referenzierte Rolle realisieren. Diese Bedingung ist in Formel 6-32 angegeben:

$$\forall t \in Template, \forall rsrs \in RequiredServiceReferenceSet:$$
$$isDependendyResolved(t, rsrs) \Longrightarrow$$
$$\forall ps \in uses(rsrs): \begin{bmatrix} \left(requires\left(assignedTo(rsrs)\right)\right) \neq \emptyset \Longrightarrow \\ requires\left(assignedTo(rsrs)\right).isConformTo(ps) \end{bmatrix}$$

Formel 6-32

Ein *RequiredTemplateInterface* legt abschließend fest, welche *ProvidedServices* ein *RequiredServiceReferenceSet* nutzen darf. Hierbei handelt es sich um solche Dienste, welche von einer Komponente angeboten werden, die durch genau das *Template* in die Menge *selectedComponents* aufgenommen wurde, welches mittels der Assoziation *connectedTo* referenziert ist. Diese Bedingung ist in Formel 6-33 angegeben:

$$\forall t \in Template, \forall rsrs \in RequiredServiceReferenceSet:$$
$$isDependendyResolved(t, rsrs) \Longrightarrow$$
$$\forall ps \in uses(rsrs): belongsTo(ps) \in$$
$$selectedComponents\left(offeredBy\left(connectedTo\left(assignedTo(rsrs)\right)\right)\right)$$

Formel 6-33

Hiermit ist die Definition des Prädikats *isDependencyResolved* und somit auch die des Prädikats *isValidConfiguration* aus Formel 6-17 vollständig.

Wie bereits erwähnt, definiert die Klasse *Application* einen Zustandsautomaten bestehend aus den Zuständen NOT_RUNNING und RUNNING. Mit Hilfe des Prädikats *isValidConfiguration* können nun Invarianten formuliert werden, die jeweils gelten müssen, wenn sich die Anwendung in dem einen oder dem anderen Zustand befindet. Diese Invarianten sind in Formel 6-34 angegebenen:

$$\forall app \in Application:$$
$$state(app) = NOT\_RUNNING \Leftrightarrow \neg isValidConfiguration(app) \wedge$$
$$state(app) = RUNNING \Leftrightarrow isValidConfiguration(app)$$

Formel 6-34

Eine Anwendung befindet sich demnach genau dann im Zustand NOT_RUNNING, wenn das Prädikat *isValidConfiguration* zu *false* ausgewertet wird. Ansonsten befindet er sich im Zustand RUNNING.

Aufgabe eines Frameworks ist es nun, unter Verwendung aller zur Verfügung stehenden Anwendungskomponenten eine Anwendungskonfiguration zu erzeugen, so dass das Prädikat

isValidConfiguration zu *wahr* ausgewertet wird. Ein Beispiel für einen solchen Konfigurationsalgorithmus wird im folgenden Abschnitt skizziert.

6.4 Konfigurationsprozess einer Anwendung

In diesem Abschnitt wird nun vorgestellt, wie eine Anwendungskonfiguration automatisiert erzeugt werden kann, die die zuvor definierten Anforderungen erfüllt und somit dazu führt, dass der Zustand der Anwendung vom Zustand NOT_RUNNING in den Zustand RUNNING wechselt. Im hier vorgeschlagenen Verfahren handelt es sich um einen Brute-Force-Ansatz, bei dem iterativ alle möglichen Konfigurationen erzeugt werden. Nachdem jeweils eine Konfiguration erzeugt wurde, wird überprüft, ob für diese das Prädikat *isValidConfiguration* zu wahr ausgewertet werden kann.

6.4.1 Eine mögliche Realisierung des Zustandsautomaten von *Application*

Die Bedingungen, unter denen sich eine Anwendung im Zustand NOT_RUNNING bzw. RUNNING befinden darf, wurden in Formel 6-34 im vorangegangenen Abschnitt definiert. Ein möglicher Zustandsautomat, der diese Bedingung berücksichtigt, ist in Abbildung 6-12 dargestellt.

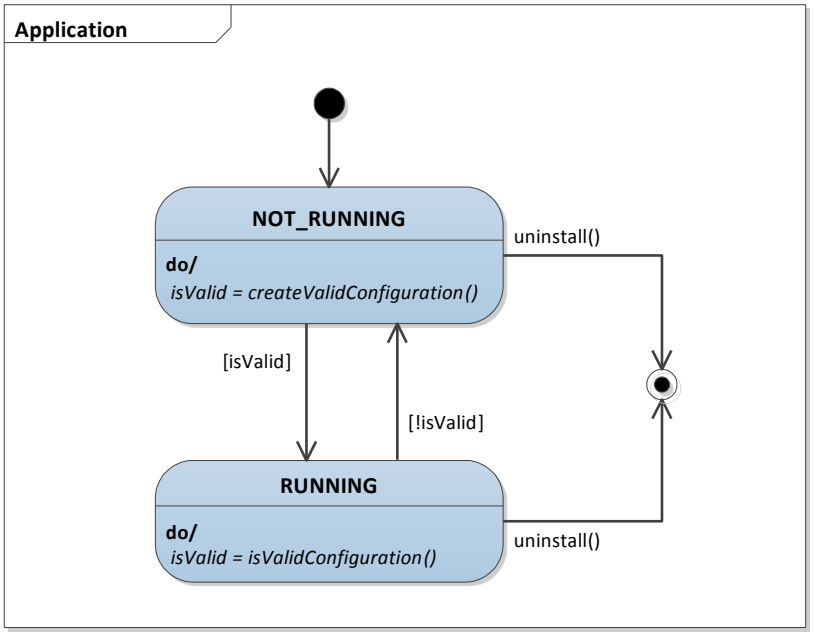


Abbildung 6-12: Der Zustandsautomat von *Application*

Zu Beginn befindet sich der Automat einer Anwendung im Zustand NOT_RUNNING. Innerhalb dieses Zustandes werden iterativ Anwendungskonfigurationen erzeugt und jeweils überprüft, ob das Prädikat *isValidConfiguration* (siehe Formel 6-17) zu *wahr* ausgewertet wird. Beides geschieht innerhalb der Methode *createValidConfiguration,* welche in Listing 6-2 in Form von Pseudocode angegeben ist. Die einzelnen Elemente des Algorithmus werden in den folgenden Abschnitten noch etwas genauer erläutert.

```
1  boolean createValidConfiguration() {
2    while(possibleComponentAssignmentSets.hasNext()) {
3      possibleComponentAssignmentSets.next().realize();
4
5      while(possibleInterfaceAssignmentSets.hasNext()) {
6         possibleInterfaceAssignmentSet.next().realize();
7
8        while(possibleUsageSets.hasNext()) {
9          possibleUsageSets.next().realize();
10
11         if(isValidConfiguration()) {
12           return true;
13         }
14       }
15     }
16   }
17   return false;
18 }
```

Listing 6-2: Algorithmus zur Erzeugung einer gültigen Anwendungskonfiguration

Sobald eine gültige Konfiguration erzeugt werden konnte, wechselt der Automat in den Zustand RUNNING. Auf Grund dessen, dass eine gültige, den Anforderungen entsprechende Konfiguration sich jederzeit derart verändern kann, so dass sie den Anwendungsarchitektur-spezifischen Anforderungen nicht mehr gerecht wird, wird eine zyklische Überprüfung der Konfiguration innerhalb des Zustandes durchgeführt. Sobald die Konfiguration nicht mehr den definierten Anwendungsarchitektur-spezifischen Anforderungen entspricht und somit das Prädikat *isValidConfiguration* zu *false* ausgewertet wird, wechselt der Automat zurück in den Zustand NOT_RUNNING. Bei dem hier vorgegebenen Algorithmus handelt es sich lediglich um eine von zahlreichen Möglichkeiten der Realisierung des Automaten.

Hauptaufgabe des Konfigurationsprozesses ist es also, bei gegebener Menge von Komponenten eine Anwendungskonfiguration zu erzeugen, die alle Bedingungen erfüllt, damit die Anwendung in den Zustand RUNNING übergehen kann. Bei dem hier vorgestellten Algorithmus handelt es sich, wie bereits zuvor erwähnt, um einen Brute-Force-Ansatz. Zu diesem Zweck erzeugt das Framework zunächst eine Konfiguration, die alle strukturellen Anforderungen erfüllt. Diese Konfiguration wird ausgeführt und die Dienste gestartet. Erst im nächsten Schritt wird dann überprüft, ob sämtliche Anforderungen an den Zustand der Dienste erfüllt werden, da diese Anforderungen erst dann überprüft werden können, wenn die Dienste ausgeführt werden. Werden die Anforderungen nicht erfüllt, muss eine neue strukturell kompatible Konfiguration erzeugt werden.

Der Algorithmus gliedert sich in zwei Teile, nämlich zum einen in den Teil, der eine Anwendungskonfiguration erzeugt (Zeilen 2-9), und in den Teil, der diese Konfiguration auf Konformität zu den Anforderungen überprüft (Zeilen 11-13). Zur Erzeugung einer Konfiguration sind wiederum drei Schritte nötig. Zunächst muss eine Auswahl an Komponenten getroffen werden. Anschließend müssen die *ProvidedServices* und *RequiredServiceReferenceSets* jeweils einem *ProvidedTemplateInterface* bzw. *RequiredTemplateInterface* zugeordnet werden. Es müssen also die beiden *assignedTo*-Mengen definiert werden. Und schließlich muss die Menge *uses* für jedes *RequiredServiceReferenceSet* bestimmt werden. In jeder der drei Phasen gibt es in der Regel zahlreiche Variationen, wobei im ungünstigsten Fall sämtliche Varianten erzeugt und

getestet werden müssen. In den folgenden Abschnitten werden nun die einzelnen Phasen der Konfigurationserzeugung näher erläutert.

6.4.2 Auswahl von Komponenten für eine Anwendungskonfiguration

Die Ausgangssituation des Konfigurationsprozesses ist eine Menge zur Verfügung stehender Komponenten. Für eine Anwendungskonfiguration muss hieraus eine Auswahl getroffen werden. Hierzu wird für jedes *Template* eine Belegung der Menge *selectedComponents* erzeugt, wobei hier die statischen Eigenschaften einer Komponente bereits berücksichtigt werden können. Die Anwendung berechnet auf Basis der zur Verfügung stehenden Komponenten und der Anwendungsspezifikation die Menge aller möglichen Belegungskombinationen und stellt diese über einen Iterator zur Verfügung (*possibleComponentAssignmentSets* aus Listing 6-2). Die Methode *realize* setzt die konkrete Belegung um.

Zur Verdeutlichung sei eine Anwendungsspezifikation mit zwei *Templates* gegeben, welche in Abbildung 6-13 dargestellt ist. Außerdem sei angenommen, dass fünf Komponenten im System zur Verfügung stehen.

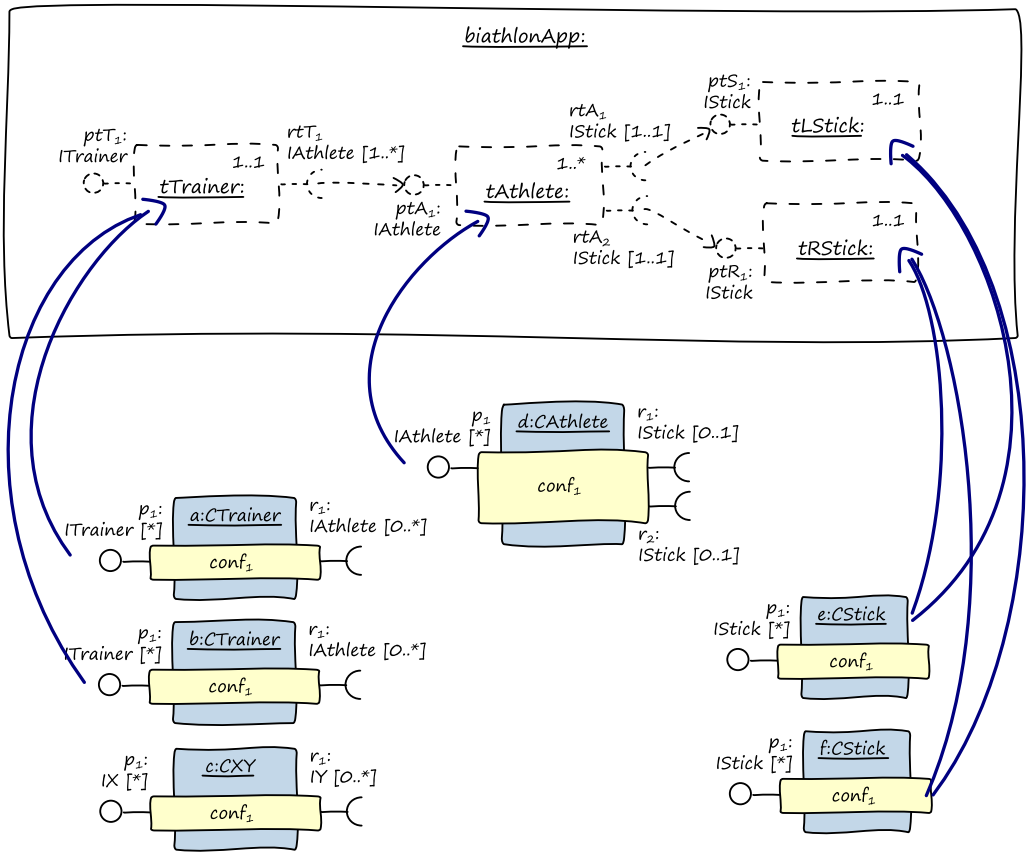


Abbildung 6-13: Beispiel für mögliche Zuordnungen von Komponenten zu Templates

In diesem Beispiel können die Komponenten *a* und *b* dem *Template tTrainer* zugeordnet werden, wobei diesem *Template* genau eine Komponente zugeordnet werden muss, um die Anforderungen der Anwendung zu erfüllen. Beide Komponenten bieten einen Dienst an, der die Domänenschnittstelle *ITrainer* implementiert und definieren ein *RequiredServiceReferenceSet*, welches die Domänenschnittstelle *IAthlete* referenziert. Dem *Template tAthlete* kann lediglich die Komponente *d* zugeordnet werden, da diese Komponente die einzige ist, die die

strukturellen Anforderungen des *Templates* erfüllt. Für die *Templates tLStick* und *tRStick* stehen insgesamt zwei Komponenten zur Verfügung, wobei jedem dieser beiden *Templates* genau eine Komponente zugeordnet werden muss, um die Anforderungen der Anwendung erfüllen zu können. Hieraus ergeben sich nun folgende mögliche Zuordnungen von Komponenten zu *Templates*:

$$possibleComponentAssignmentSet_1 = \begin{bmatrix} selectedComponents(tTrainer) = \{a\}, \\ selectedComponents(tAthlete) = \{d\}, \\ selectedComponents(tLStick) = \{e\}, \\ selectedComponents(tRStick) = \{f\} \end{bmatrix}$$

$$possibleComponentAssignmentSet_2 = \begin{bmatrix} selectedComponents(tTrainer) = \{b\}, \\ selectedComponents(tAthlete) = \{d\}, \\ selectedComponents(tLStick) = \{e\}, \\ selectedComponents(tRStick) = \{f\} \end{bmatrix}$$

$$possibleComponentAssignmentSet_3 = \begin{bmatrix} selectedComponents(tTrainer) = \{a\}, \\ selectedComponents(tAthlete) = \{d\}, \\ selectedComponents(tLStick) = \{f\}, \\ selectedComponents(tRStick) = \{e\} \end{bmatrix}$$

$$possibleComponentAssignmentSet_4 = \begin{bmatrix} selectedComponents(tTrainer) = \{b\}, \\ selectedComponents(tAthlete) = \{d\}, \\ selectedComponents(tLStick) = \{f\}, \\ selectedComponents(tRStick) = \{e\} \end{bmatrix}$$

Innerhalb des Algorithmus aus Listing 6-2 wird nun mit Hilfe des Iterators *possibleComponentAssignmentSets* über diese Möglichkeiten iteriert (Zeile 2+3). Hierbei wird dann die zurückgelieferte Belegung durch Aufruf von *realize* durch das Framework umgesetzt. Im nächsten Schritt folgt dann die Zuordnung von *ProvidedServices* zu *ProvidedTemplateInterfaces* und *RequiredServiceReferenceSets* zu *RequiredTemplateInterfaces* (siehe nächster Abschnitt).

6.4.3 Zuordnung von Komponentenschnittstellen zu Templateschnittstellen

Es kann vorkommen, dass *ProvidedServices* einer Komponente mehreren *ProvidedTemplateInterfaces* zugeordnet werden können. Da zur Laufzeit *ProvidedServices* den *ProvidedTemplateInterfaces* zugeordnet werden müssen, muss das Framework hier eine Auswahl treffen. Selbiges gilt für *RequiredServiceReferenceSets* und *RequiredTemplateInterfaces*. Zur Verdeutlichung sei angenommen, dass die *RequiredTemplateInterfaces* des *Templates tAthlete* aus Abbildung 6-13 keine Schnittstellenrollen referenzieren, sondern lediglich die Domänenschnittstelle *IStick*, wie in Abbildung 6-14 dargestellt.

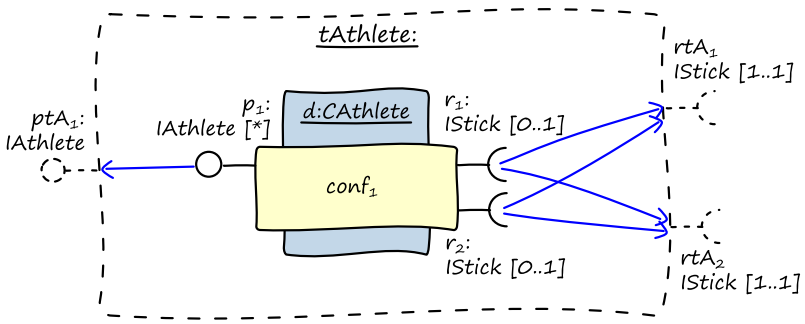


Abbildung 6-14: Beispiel für mögliche Zuordnungen von Komponentenschnittstellen zu Templateschnittstellen

In diesem Beispiel kann das *RequiredServiceReferenceSet* r_1 sowohl dem *RequiredTemplateInterface* rtA_1 als auch rtA_2 zugeordnet werden. Selbiges gilt für das *RequiredServiceReferenceSet* r_2.

Innerhalb des Algorithmus aus Listing 6-2 wird nun über alle möglichen Zuordnungen iteriert, welche sich aus der Zuordnung von Komponenten zu *Templates* im vorangegangenen Schritt ergeben. Im Modell ist die Zuordnung durch die Assoziation *assignedTo* zwischen *RequiredServiceReferenceSet* und *RequiredTemplateInterface* bzw. zwischen *ProvidedService* und *ProvidedTemplateInterface* repräsentiert. Die möglichen Belegungen, welche sich für das Beispiel aus Abbildung 6-14 ergeben sind im Folgenden dargestellt:

$$possibleInterfaceAssignmentSet_1 = \begin{bmatrix} assignedTo(p_1) = ptA_1 \\ assignedTo(r_1) = rtA_1 \\ assignedTo(r_2) = rtA_2 \end{bmatrix}$$

$$possibleInterfaceAssignmentSet_2 = \begin{bmatrix} assignedTo(p_1) = ptA_1 \\ assignedTo(r_1) = rtA_2 \\ assignedTo(r_2) = rtA_1 \end{bmatrix}$$

Innerhalb des Algorithmus aus Listing 6-2 wird nun mit Hilfe des Iterators *possibleInterfaceAssignmentSets* über diese Möglichkeiten iteriert (Zeile 5+6). Hierbei wird dann die zurückgelieferte Belegung durch Aufruf von *realize* durch das Framework umgesetzt. Im nächsten Schritt folgt dann die Belegung der Menge *uses* für die *RequiredServiceReferenceSets* der Komponenten, welche zuvor der Menge *selectedComponents* zugeordnet wurden (siehe nächster Abschnitt).

6.4.4 Belegung der Menge *uses* und Überprüfung der Konfiguration

Der letzte Schritt des Algorithmus zur Erzeugung einer Konfiguration besteht darin, die Nutzungsbeziehungen zwischen Komponenten herzustellen. Ziel ist eine Belegung der Menge *uses* für jedes *RequiredServiceReferenceSet* aller an der Anwendung beteiligten Komponenten, so dass die Anforderungen der Anwendungsspezifikation erfüllt werden können.

Häufig tritt hierbei der Fall auf, dass mehrere Möglichkeiten zur Belegung dieser Menge bestehen. Zur Illustration sei folgende Situation angenommen (siehe Abbildung 6-15):

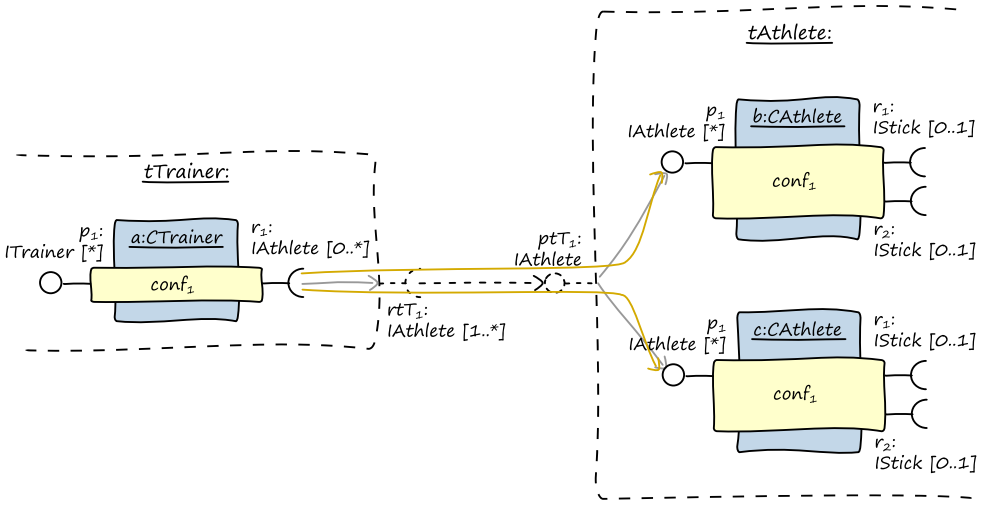


Abbildung 6-15: Beispiel für mögliche Belegungen der Menge *uses*

In diesem Fall kommt für das *RequiredServiceReferenceSet* r_1 der Trainerkomponente der angebotene Dienst beider Athletenkomponenten zur Nutzung in Frage. Entsprechend ergeben sich die im Folgenden angegebenen Belegungen der Menge *uses*:

$$possibleUsageSet_1 = [uses(r_1\_a) = \{p_1\_b\}]$$
$$possibleUsageSet_2 = [uses(r_1\_a) = \{p_1\_c\}]$$
$$possibleUsageSet_3 = [uses(r_1\_a) = \{p_1\_b, p_1\_c\}]$$

Die leere Menge wäre in diesem Fall keine gültige Belegung, da im Attribut *minNoOfRequiredRefs* der Komponente der Wert 1 hinterlegt ist. Im Algorithmus aus Listing 6-2 wird mit Hilfe des Iterators *possibleUsageSets* über diese möglichen Belegungen iteriert um gültige Anwendungskonfigurationen zu erzeugen.

Nachdem eine Komponentenauswahl getroffen wurde, anschließend eine Zuordnung der Schnittstellen stattgefunden hat und abschließend die Menge *uses* aller *RequiredServiceReferenceSets* belegt wurde, entsteht automatisch eine laufende Konfiguration. Hierzu wird den selbstkonfigurierenden Komponenten in jedem Konfigurationsschritt mitgeteilt, ob sie Teil der Anwendung sind, welchem *Template* sie sich zuordnen sollen, welchen *Template-Schnittstellen* dessen Dienste und *RequiredServiceReferenceSets* zugeordnet werden sollen und mit welchen Diensten sie sich verbinden sollen. Die einzelnen Iteratoren des Algorithmus werden hierzu von den einzelnen Komponenten realisiert. Auf diese Weise benötigt die Applikation keine Übersicht darüber, welche Komponente gerade mit welcher anderen Komponente verbunden ist. Sie muss lediglich die Methode *next* des jeweiligen Iterators an den entsprechenden Stellen des Algorithmus aufrufen und sicherstellen, dass der gesamte Lösungsraum durch abgedeckt wird.

Nach Erzeugung einer Konfiguration mit dem zuvor vorgestellten Verfahren können jetzt auch die übrigen Anforderungen der Anwendungsspezifikation auf Konformität überprüft werden. Hierzu muss nun das Prädikat *isValidConfiguration* ausgewertet werden (siehe Formel 6-17). Erst, wenn auch dieses Prädikat zu *wahr* ausgewertet wird, wechselt die Anwendung in den Zustand RUNNING. Andernfalls muss eine neue Konfiguration erzeugt werden, indem eine der zuvor beschriebenen Variationen realisiert wird.

Bei dem hier angegebenen Algorithmus handelt es sich lediglich um eine Skizzierung dessen, wie eine Konfiguration erzeugt werden kann, welche mit den definierten Anwendungsarchitektur-spezifischen Anforderungen konform ist. In der Zusammenfassung dieser Arbeit werden weitere Möglichkeiten aufgezeigt, auf welche Weise dieser Prozess ggf. optimiert werden könnte.

6.5 Beispiel eines Konfigurationsprozesses unter Berücksichtigung anwendungsspezifischer Anforderungen

In diesem Abschnitt wird nun anhand eines Beispiels beschrieben, wie mit Hilfe des soeben skizzierten Konfigurationsalgorithmus automatisiert eine Konfiguration erzeugt werden kann, die den Anwendungsarchitektur-spezifischen Anforderungen entspricht. Hierzu wird die bereits in Abschnitt 6.2.5 vorgestellte Anwendungsarchitektur-Spezifikation verwendet, welche in Abbildung 6-16 nochmals dargestellt ist.

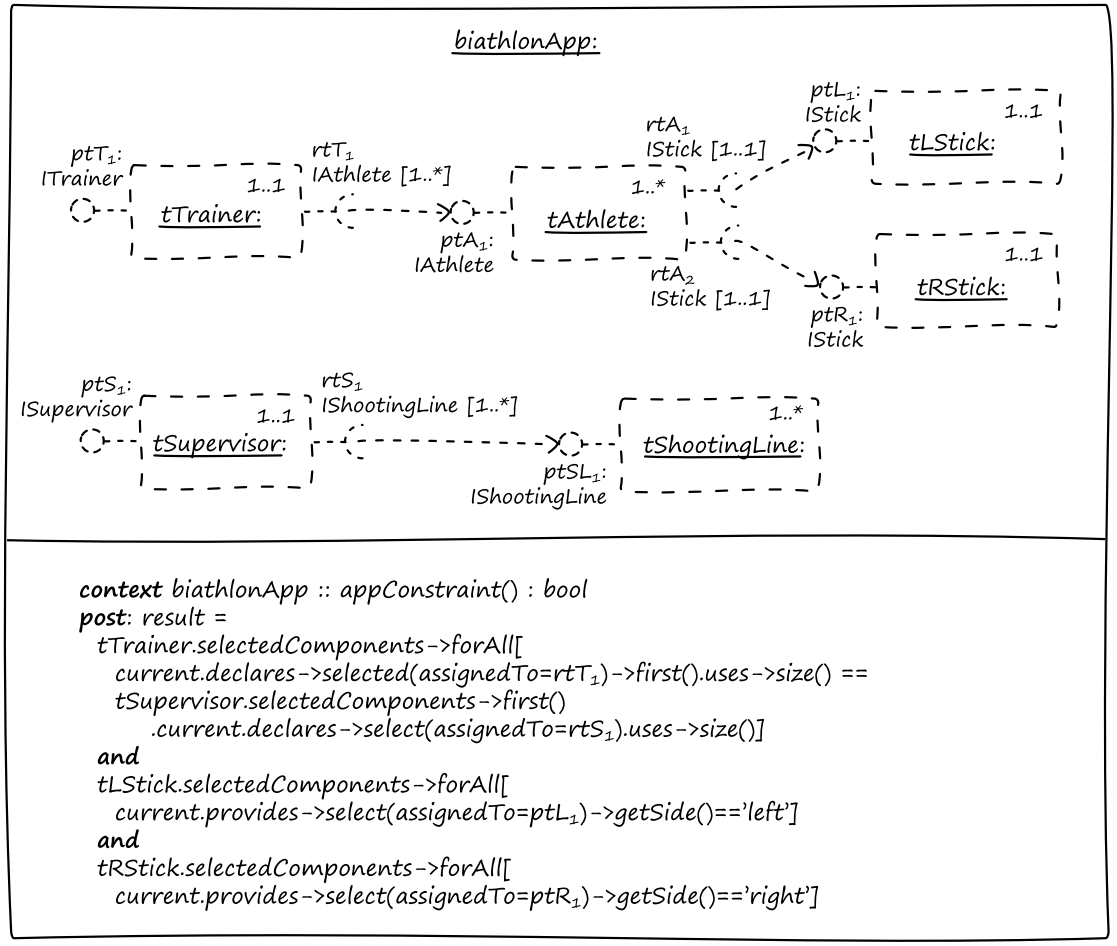


Abbildung 6-16: Spezifikation einer Anwendung zur Trainingsunterstützung für Biathleten

Diese Anwendung realisiert im Wesentlichen ein Techniktraining, welches mit Schießtraining gekoppelt ist. Hierzu werden für die Anwendung insgesamt sechs *Templates* definiert, welche Platzhalter für Trainer-, Athleten-, Skistock-, Schießaufsichts- sowie Schießstandkomponenten darstellen. Zur Durchführung der Schießübungen muss hierbei sichergestellt werden, dass die Schießaufsichtkomponente mit genauso vielen Schießstandkomponenten verbunden ist, wie Sportlerkomponenten mit der Trainerkomponente verbunden sind. Diese Anforderung ist mit

Hilfe des Prädikats *appConstraint* der Klasse *Application* definiert und im unteren Teil der Abbildung in Form eines OCL-Constraints angegeben. Des Weiteren sei angenommen, dass die in Abbildung 6-17 dargestellte Menge von Komponenten im System zur Verfügung steht.

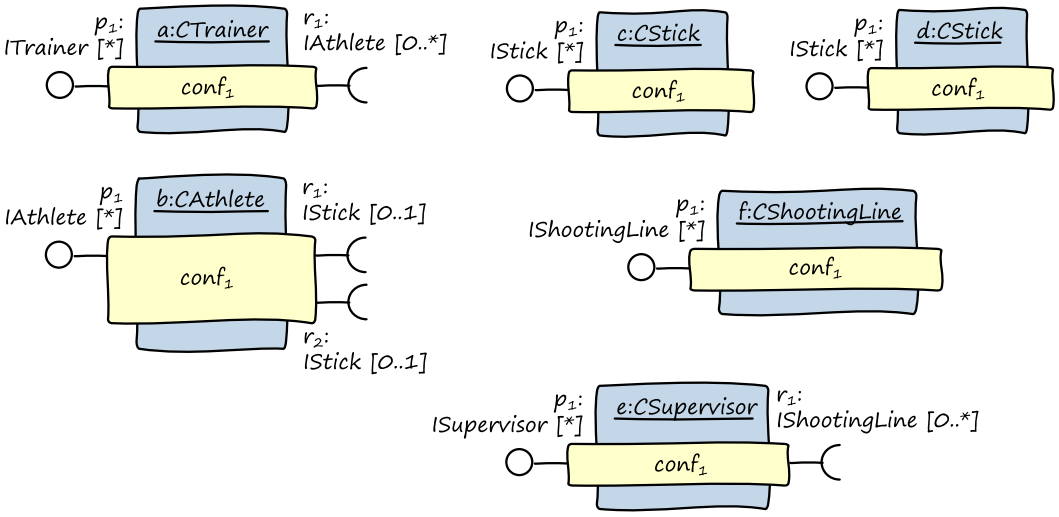


Abbildung 6-17: Komponenten des Beispiels

Im ersten Schritt wird nun zunächst eine erste Zuordnung von Komponenten zu *Templates* vorgenommen, so dass die strukturellen Anforderungen erfüllt sind (siehe Abbildung 6-18).

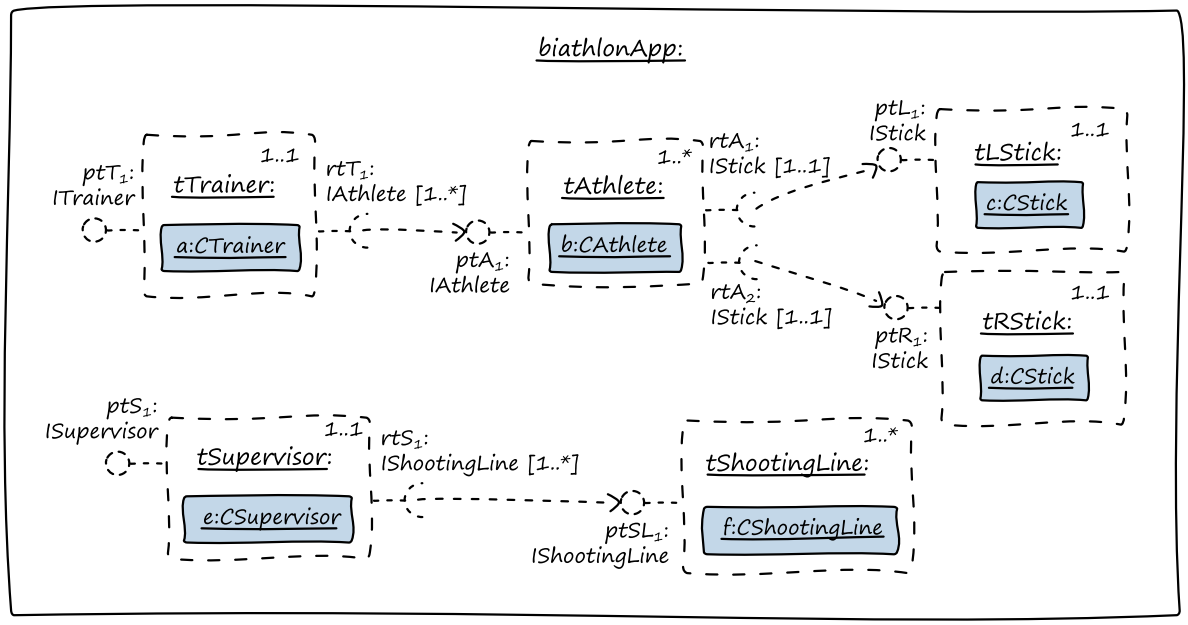


Abbildung 6-18: Zuordnung von Komponenten zu *Templates* auf Basis der Komponentenstruktur

Es handelt sich hierbei um eine von mehreren möglichen Belegungen der Menge *selectedComponents*, welche bereits die Anforderungen an die Struktur einer Komponente sowie Anforderungen bzgl. der Anzahl an benötigten Komponenten erfüllt. Eine alternative Belegung könnte beispielsweise die Komponente *c* dem *Template tRStick* zuordnen, und die Komponente *d* dem *Template tRStick*.

Anschließend wird für jede Komponente eine Zuordnung von *ProvidedServices* und *RequiredServiceReferenceSets* zu jeweils einem *ProvidedTemplateInterface* bzw.

RequiredTemplateInterface vorgenommen. Bei der hier vorgenommenen Belegung der Menge *selectedComponents* der einzelnen *Templates* ist die Zuordnung eindeutig (siehe Abbildung 6-19).

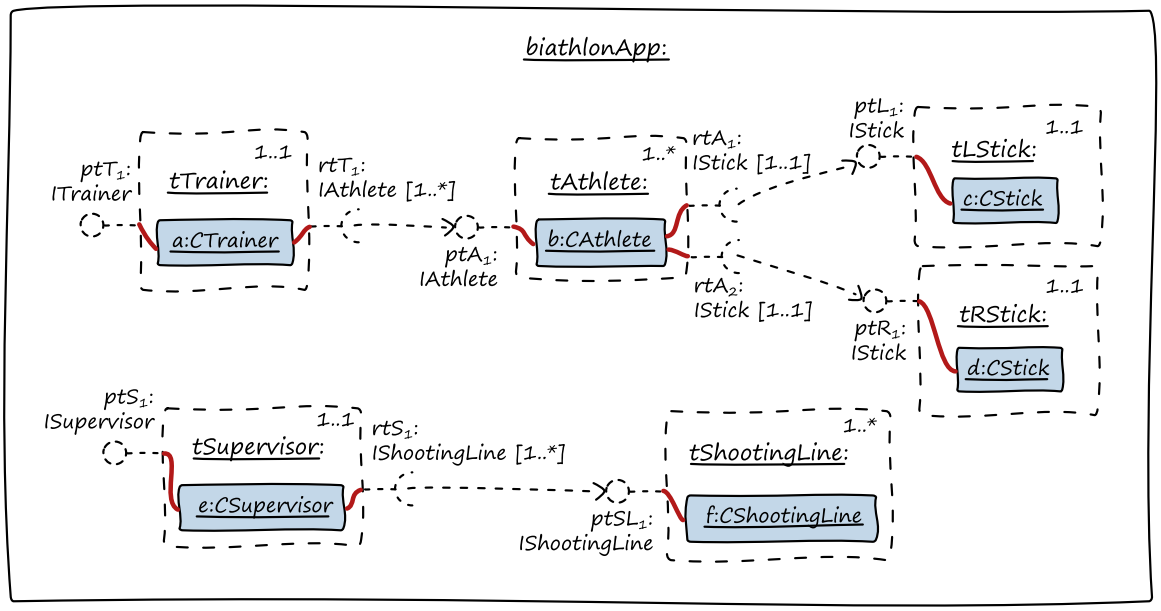


Abbildung 6-19: Zuordnung von *ProvidedServices* und *RequiredServiceReferenceSets* zu *ProvidedTemplateInterfaces* und *RequiredTemplateInterfaces*

Nachdem somit die Mengen *selectedComponents* sowie *assignedTo* belegt wurden, folgt im nächsten Schritt des Konfigurationsalgorithmus die Belegung der *uses*-Mengen der einzelnen *RequiredServiceReferenceSets* der Komponenten der Menge *selectedComponents* unter Berücksichtigung der Anforderungen der jeweiligen *RequiredTemplateInterfaces*. Eine mögliche Belegung ist in Abbildung 6-20 dargestellt.

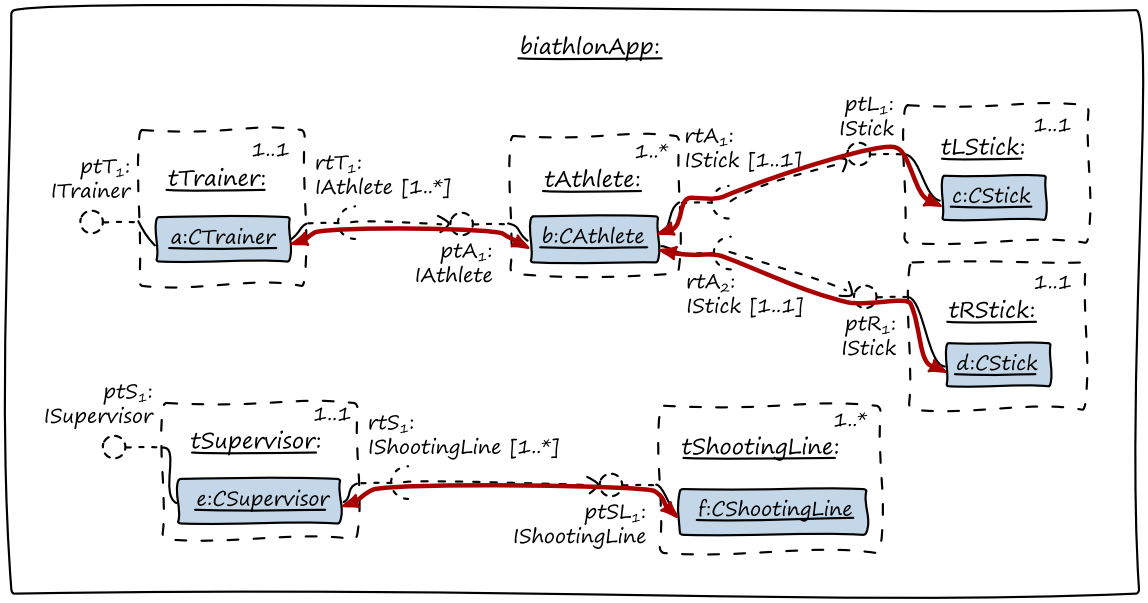


Abbildung 6-20: Eine mögliche Belegung der uses-Mengen der einzelnen Komponenten

Die einzelnen Komponenten bzw. deren angebotene Dienste werden nun ausgeführt und es entsteht eine laufende Konfiguration, welche alle Anforderungen bzgl. der Anzahl verwendeter

Komponenten als auch die Anzahl der Elemente in den *uses*-Mengen erfüllt. Bevor die Anwendung nun in den Zustand RUNNING wechseln kann, muss noch das Prädikat *appConstraint* auf Gültigkeit geprüft werden. Ist dies der Fall, so kann der entsprechende Zustandswechsel stattfinden. Ansonsten muss eine neue Konfiguration erzeugt werden, indem z.B. die Zuordnung von Komponenten zu *Templates*, die Zuordnung von Komponenten- zu *Template*-Schnittstellen oder die Belegung der *uses*-Mengen verändert wird. Ebenso muss das Framework auf Ereignisse wie das Hinzukommen oder den Wegfall von Komponenten reagieren.

6.6 Zusammenfassung

In diesem Kapitel wurden die zuvor beschriebenen Konzepte zur Konfiguration selbstorganisierender Softwaresysteme dahingehend erweitert, dass nun auch komponentenübergreifende Anforderungen berücksichtigt werden können.

Zunächst wurde hierzu das Modell erweitert um die Möglichkeit, komponentenübergreifende, Anwendungsarchitektur-spezifische Anforderungen angeben zu können, welches einen Hauptbeitrag dieser Arbeit darstellt. Hierbei sieht die Lösung vor, diese Anforderungen definieren zu können, ohne weder die konkreten Komponenteninstanzen noch die Komponententypen kennen zu müssen. Es können hierbei zur Laufzeit neue Komponententypen und Instanzen automatisiert in die Anwendung integriert werden, ohne diese zum Zeitpunkt der Spezifikation der Anwendungsarchitektur-spezifischen Anforderungen kennen zu müssen. Dies ermöglicht die Erhaltung emergenten Systemverhaltens, wie es in selbstorganisierenden Systemen meist gefordert wird.

Des Weiteren wurde in diesem Kapitel ein möglicher Algorithmus zur Erzeugung einer Anwendungsarchitektur-konformen Konfiguration vorgeschlagen. Dieser basiert darauf, auf Basis der Menge aller verfügbaren Komponenten eine Konfiguration nach der anderen zu erzeugen und jede erzeugte Konfiguration auf Konformität zu prüfen. Sobald eine entsprechende Konfiguration gefunden wurde, wechselt die Anwendung in den Zustand RUNNING und wird ausgeführt.

7 Implementierung und Anwendung des Frameworks

In den vorangegangenen Kapiteln wurde ein Komponentenmodell vorgestellt, welches die Konfiguration dynamisch-adaptiver Anwendungen auf Basis selbstkonfigurierender Komponenten ermöglicht. Hierbei wurden in Kapitel 4 und 5 zunächst die Grundlagen gelegt und ein Modell eingeführt, welches zunächst die Berücksichtigung anwendungsspezifischer Anforderungen außer Acht gelassen hat. Im Kapitel 6 wurde das Modell dann entsprechend erweitert um die Möglichkeit, anwendungsspezifische Anforderungen spezifizieren und automatisiert umsetzen zu lassen. In diesem Kapitel wird nun eine prototypische Implementierung der zuvor vorgestellten Konzepte erläutert, welche im Rahmen dieser Arbeit realisiert wurde. Hierzu werden im folgenden Abschnitt zunächst die Klassen und Schnittstellen des Frameworks erläutert und anschließend deren Verwendung zur Entwicklung dynamisch-adaptiver Anwendungen anhand eines Beispiels gezeigt.

7.1 Klassen und Schnittstellen des Frameworks

Basis dieses Frameworks bildet eine Reihe von Schnittstellen, welche die einzelnen Elemente des zuvor vorgestellten Komponentenmodells repräsentieren. Diese Schnittstellen und deren Assoziationen untereinander sind in Abbildung 7-1 dargestellt.

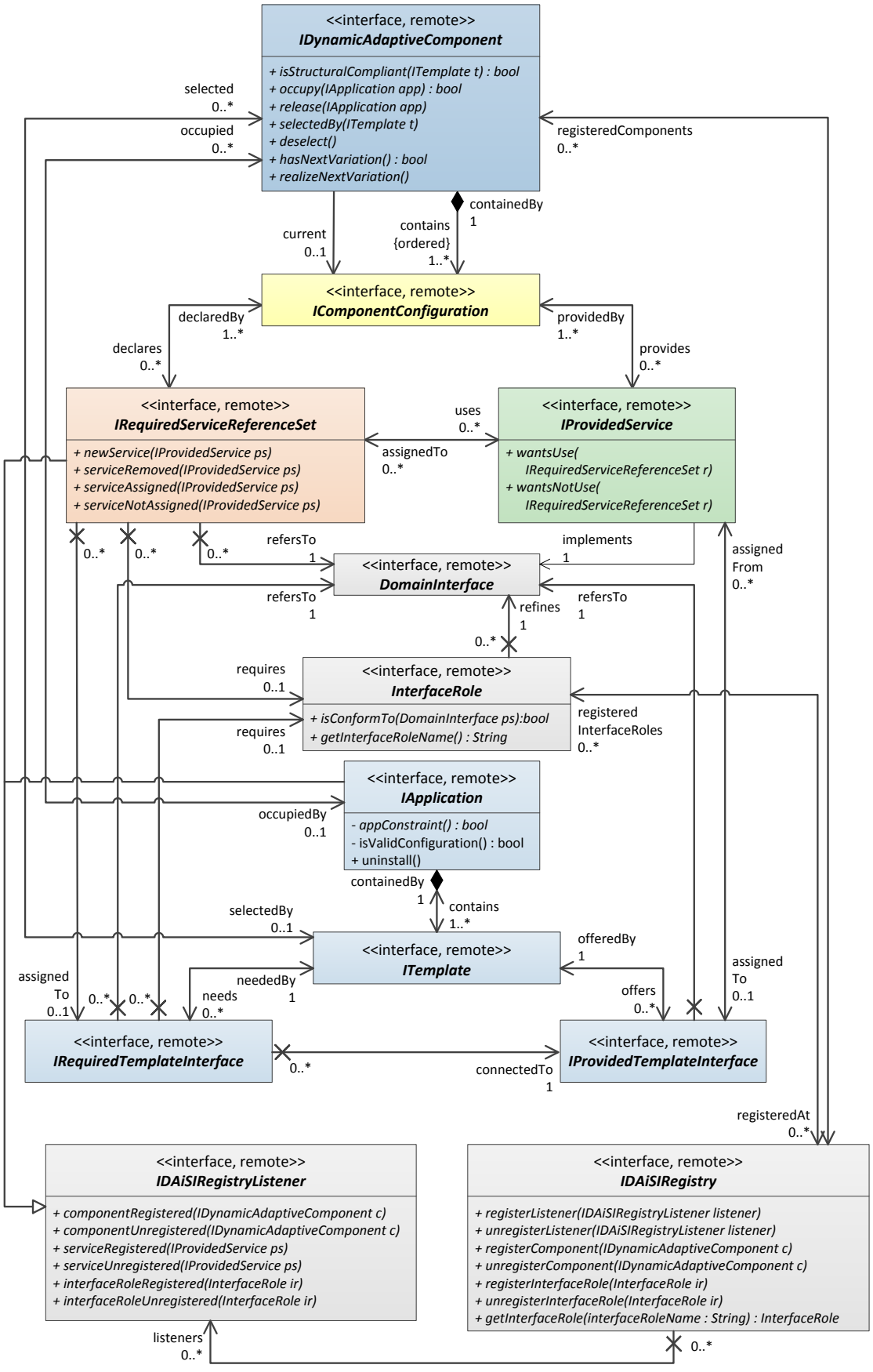


Abbildung 7-1: Die Schnittstellen des Frameworks

Jedes Element des zuvor vorgestellten Komponentenmodells ist innerhalb des Frameworks durch eine Schnittstelle realisiert. Neben diesen Schnittstellen stellt das Framework auch eine Implementierung dieser Schnittstellen zur Verfügung, welche später zur Komponenten- und Anwendungsentwicklung verwendet werden können.

Neben der Realisierung des Komponentenmodells beinhaltet das Framework eine sogenannte *Registry*. Diese dient auf der einen Seite dazu, Komponenten und deren angebotene Dienste, sowie Schnittstellenrollen im System bekannt zu machen. Auf der anderen Seite können diese Komponenten, Dienste und Schnittstellenrollen über die *Registry* von anderen Komponenten sowie Anwendungen gefunden werden. Die *Registry* wird in Abschnitt 7.1.1 vorgestellt.

Die gesamte Infrastruktur wurde auf Basis der Programmiersprache Java [Ull11] und RMI (Remote Method Invocation) entwickelt [Abt07]. Mit Hilfe von RMI lassen sich auf einfache Weise verteilte Java-Programme entwickeln, bei denen einzelne Programmbestandteile auf unterschiedlichen Rechnern und unterschiedlichen virtuellen Maschinen ausgeführt werden können. Wesentlicher Bestandteil ist die Registry. Hier können zum einen Referenzen auf Objekte unter einem aussagekräftigen Namen veröffentlicht werden. Zum anderen können andere Java-Objekte diese Referenzen mit Hilfe des namens finden und auf diese dann zugreifen, als handelte es sich um Referenzen auf lokale Java-Objekte.

Im Folgenden werden nun die einzelnen Bestandteile der Infrastruktur, im weiteren Verlauf DAiSI (Dynamic Adaptive System Infrastructure) genannt, etwas detaillierter vorgestellt. Es handelt sich hierbei um eine Neuimplementierung der in [NKA+07] vorgestellten Infrastruktur auf Basis der hier vorgestellten Konzepte. Anschließend wird anhand eines Beispiels gezeigt, wie diese Infrastruktur verwendet werden kann, um selbstorganisierende Systeme zu entwickeln und auszuführen.

7.1.1 Registry und Repository

Die Registry der Infrastruktur ermöglicht die Veröffentlichung und Bereitstellung von Komponenten, angebotenen Diensten sowie Schnittstellenrollen. Zu diesem Zweck definiert das Framework die Schnittstelle *IDAiSIRegistry*. Realisiert wird die Schnittstelle innerhalb der Infrastruktur durch die Klasse *DAiSIRegistry* (siehe Abbildung 7-2).

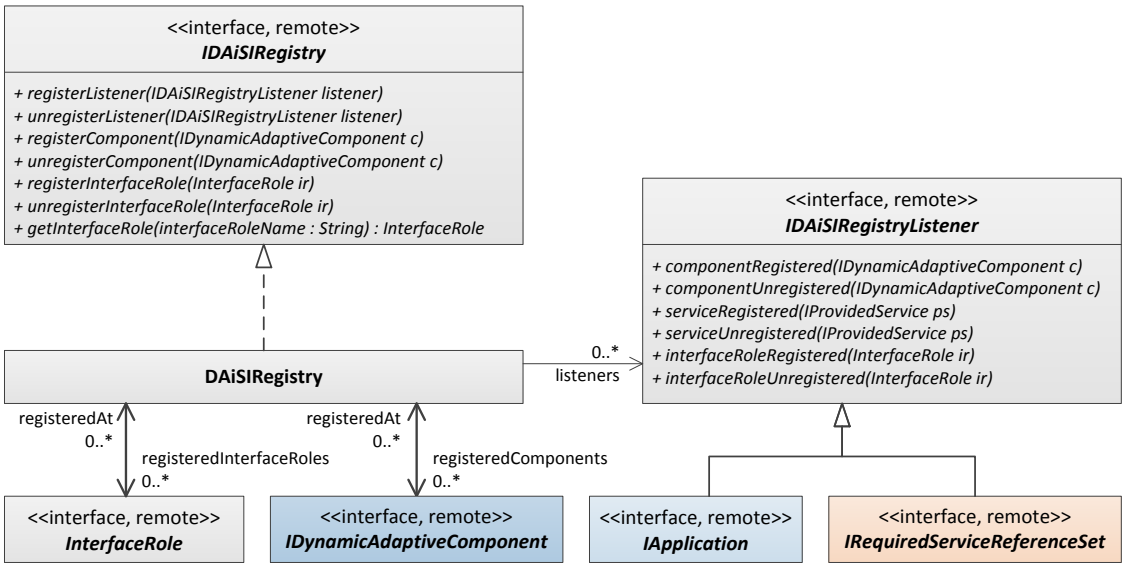


Abbildung 7-2: Realisierung der Schnittstellen *IDAiSIRegistry* und *IDAiSIRegistryListener.*

Die Schnittstelle *IDAiSIRegistry* definiert im Wesentlichen Methoden zur Registrierung und Entfernung von Referenzen auf Komponenten, Dienste und Schnittstellenrollen. Innerhalb der Infrastruktur sind verschiedene Akteure an Informationen bzgl. verfügbaren Komponenten, Diensten und Schnittstellenrollen interessiert. Um diese Akteure über entsprechende Veränderungen zu informieren, wurde das Entwurfsmuster *Beobachter* verwendet [GHJ94]. Die *Registry* übernimmt in diesem Fall die Rolle des *Observables*, alle anderen Akteure die des *Observers*. Die Schnittstelle *IDAiSIRegistryListener* muss von jedem *Observer* implementiert werden. Zusätzlich muss sich jeder *Observer* durch Aufruf der Methode *registerListener* bei der *Registry* als solcher registrieren. Die *Registry* informiert anschließend sämtliche registrierten *Observer* durch Aufruf der entsprechenden Methode der Schnittstelle *IDAiSIRegistryListener* über etwaige Veränderungen innerhalb der *Registry*.

7.1.2 DomainInterface und InterfaceRole

Zum Framework gehören zwei Schnittstellen, welche der Definition des Domänenmodells dienen. Hierzu zählt zum einen die Schnittstelle *DomainInterface*. Diese definiert keinerlei Methoden, allerdings muss jede Domänenschnittstelle von dieser Schnittstelle erben, um innerhalb der Infrastruktur als Domänenschnittstelle erkannt zu werden. Zum anderen dient die Schnittstelle *InterfaceRole* der Definition von Schnittstellenrollen. Jede Schnittstellenrolle muss diese Schnittstelle implementieren. Die Methode *isConformTo* wird später durch das Framework aufgerufen, um überprüfen zu lassen, ob der als Parameter übergebene Dienst konform zur Schnittstellenrolle ist. Mit Hilfe der Methode *getInterfaceRole* kann die Rolle erfragt werden.

7.1.3 DynamicAdaptiveComponent

Im Komponentenmodell, welches in den vorherigen Kapiteln vorgestellt wurde, repräsentiert jede Instanz der Klasse *DynamicAdaptiveComponent* die Instanz einer Komponente. Im Framework wird dieses Modellelement durch die Schnittstelle *IDynamicAdaptiveComponent* abgebildet. Die abstrakte Klasse *AbstractDynamicAdaptiveComponent* implementiert hierbei wesentliche Teile dieser Schnittstelle. Abbildung 7-3 zeigt diese Klasse gemeinsam mit den

zusätzlich benötigten Klassen und Schnittstellen. Im Folgenden werden diese detaillierter vorgestellt.

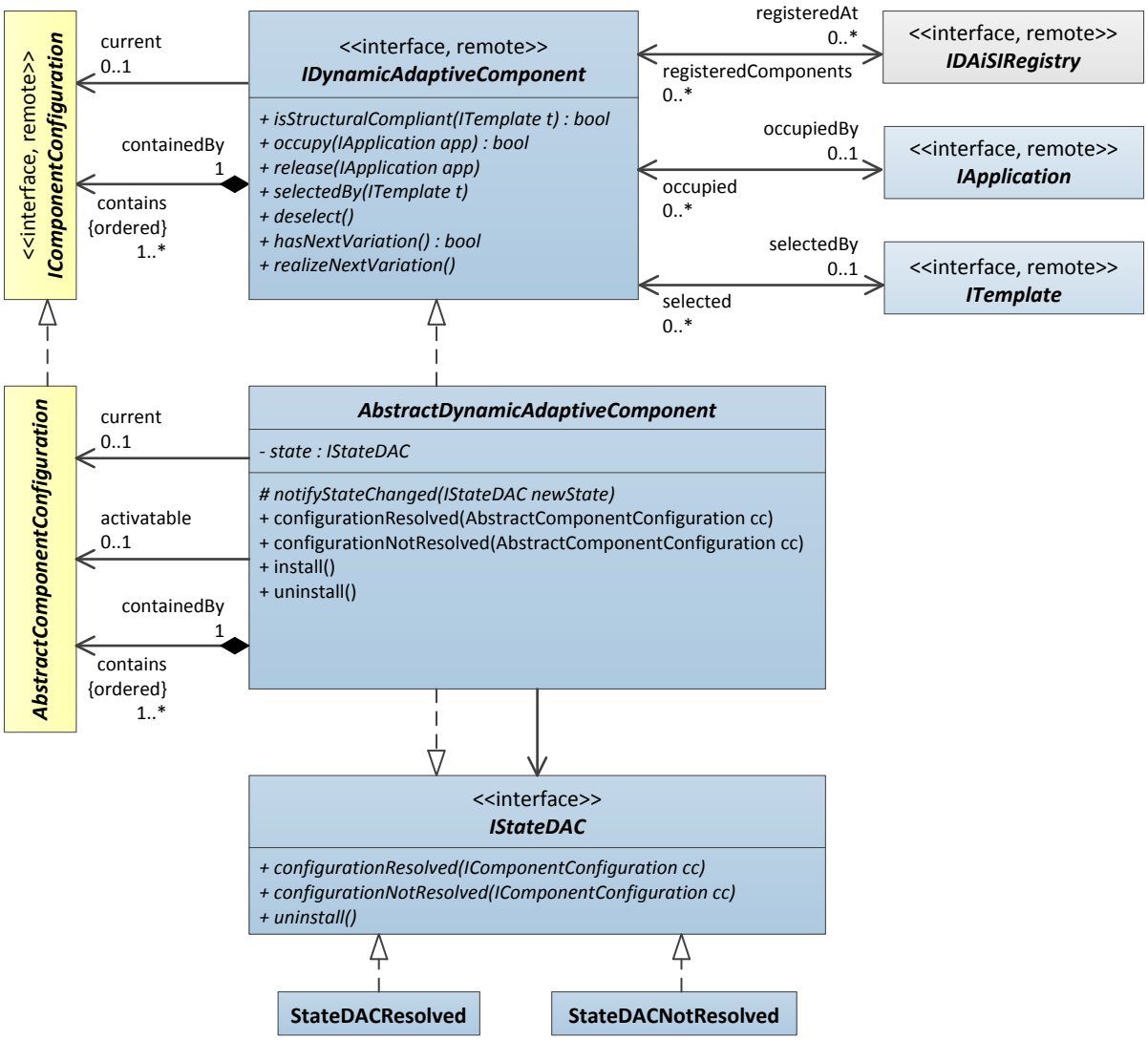


Abbildung 7-3: Implementierung der Schnittstelle I*DynamicAdaptiveComponent*

Die Klasse *AbstractDynamicAdaptiveComponent* ist im Wesentlichen verantwortlich für die Realisierung zweier Aspekte, nämlich zum einen für die Umsetzung des zuvor spezifizierten Zustandsautomaten, und zum anderen für die Umsetzung anwendungsspezifischer Anforderungen, welche durch *Templates* induziert werden.

Die Realisierung des Zustandsautomaten erfolgt mit Hilfe des Entwurfsmusters *Zustand* [GHJ94]. Hierbei definiert die Schnittstelle *IStateDAC* diejenigen Methoden, welche im Zustandsautomaten von *DynamicAdaptiveComponent* als Trigger für Zustandsübergänge dienen. Sie übernimmt innerhalb des Entwurfsmusters die Rolle *Zustand*. Die Zustände RESOLVED und NOT_RESOLVED des Zustandsautomaten werden innerhalb des Frameworks durch die Klassen *StateDACResolved* und *StateDACNotResolved* realisiert. Die Klasse *StateDACResolved* realisiert somit das Verhalten des Automaten im Zustand RESOLVED bei Aufruf eines der zwei Trigger. Ebenso implementiert die Klasse *StateDACNotResolved* das entsprechende Verhalten des Automaten im Zustand NOT_RESOLVED. Die Trigger werden hierbei zunächst bei der Klasse

AbstractDynamicAdaptiveComponent aufgerufen. Diese besitzt innerhalb der Variablen *state* eine Referenz auf diejenige Instanz, welche den aktuellen Zustand repräsentiert. Der Triggeraufruf wird durch die Klasse *AbstractDynamicAdaptiveComponent* somit lediglich an diejenige Instanz delegiert, welche in dieser Variablen hinterlegt ist.

Die zweite wesentliche Aufgabe der Klasse *AbstractDynamicAdaptiveComponent* ist die Umsetzung anwendungsspezifischer Anforderungen. Hierzu definiert das Framework die Schnittstelle *IDynamicAdaptiveComponent*. Über diese Schnittstelle kann eine Komponente einem *Template* zugeordnet werden. Außerdem können unterschiedliche Variationen bzgl. Schnittstellenzuordnung und Auswahl der verwendeten Dienste anderer Komponenten realisiert werden. Diese Schnittstelle dient somit im Wesentlichen der Realisierung des Konfigurationsalgorithmus, welcher in Abschnitt 6.4 bzw. Listing 6-2 beschrieben wurde. Über die Methode *selectedBy* wird die Komponente einem *Template* zugeordnet, während diese Zuordnung durch Aufruf von *deselect* wieder aufgehoben werden kann. Über die Methode *hasNextVariation* kann die Anwendung innerhalb des Konfigurationsalgorithmus prüfen, ob eine weitere Variation bzgl. der Zuordnung von Schnittstellen zu *Template*-Schnittstellen oder bzgl. Belegung der Menge *uses* möglich ist. Durch Aufruf von *realizeNextVariation* wird die neue Variation dann umgesetzt.

Soll nun eine neue Komponente implementiert werden, so muss diese von *AbstractDynamicAdaptiveComponent* erben und die Methode *notifyStateChanged* implementieren. Dies ist die einzige abstrakte Methode dieser abstrakten Klasse. Diese Methode wird durch das Framework immer dann aufgerufen, wenn innerhalb des Zustandsautomaten von *DynamicAdaptiveComponent* ein Zustandswechsel stattfindet. Auf diese Weise erhält der Komponentenentwickler die Möglichkeit, innerhalb seiner Komponente auf einen Zustandswechsel reagieren zu können.

Soll nun eine Komponente aus dem System entfernt werden, so muss hierzu lediglich die Methode *uninstall* der Klasse *AbstractDynamicAdaptiveComponent* aufgerufen werden. Das Framework erledigt daraufhin sämtliche Aufgaben, die für eine saubere Entfernung der Komponente aus dem System notwendig sind. Hierzu zählt insbesondere die Auflösung von Abhängigkeiten zu anderen Komponenten.

7.1.4 ComponentConfiguration

Im Komponentenmodell repräsentiert jede Instanz der Klasse *ComponentConfiguration* eine Komponentenkonfiguration. Diese Klasse ist im Framework als abstrakte Klasse namens *AbstractComponentConfiguration* realisiert. Abbildung 7-4 zeigt die Methoden und Attribute dieser Klasse sowie die zur Erbringung der Funktionalität notwendigen Hilfsklassen und Schnittstellen.

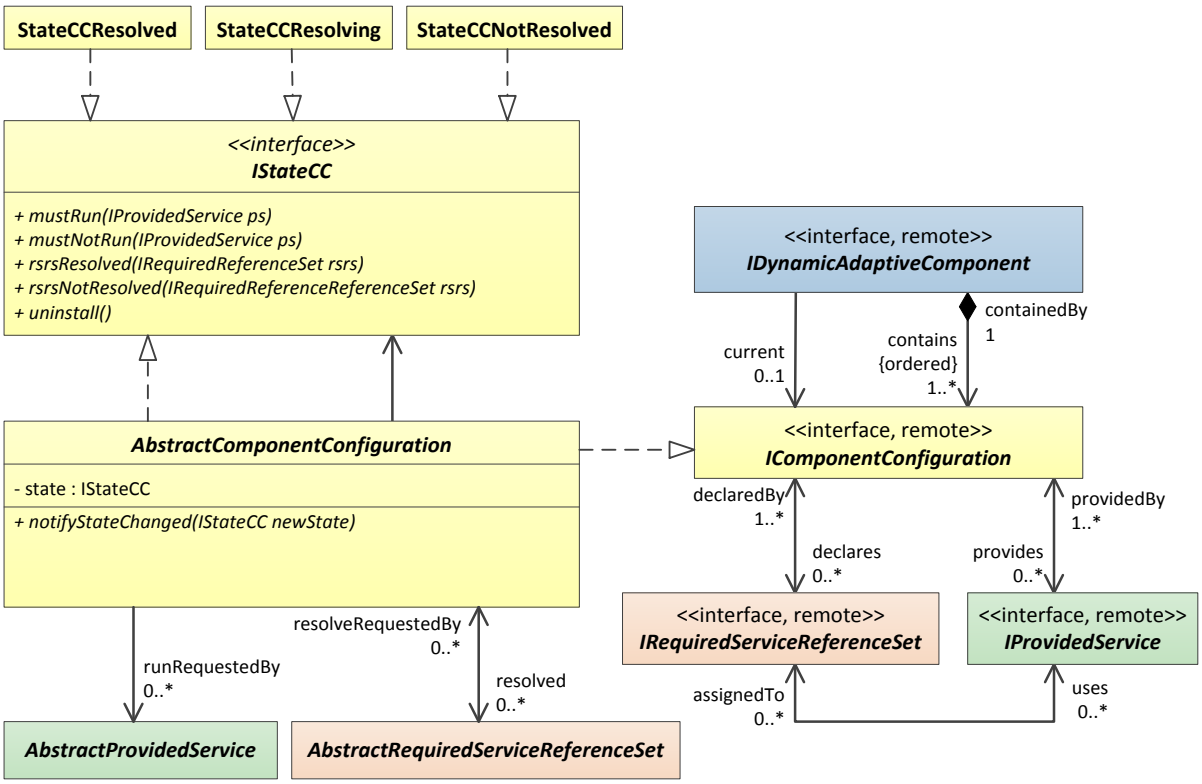


Abbildung 7-4: Realisierung der Klasse *ComponentConfiguration* des Komponentenmodells

Die Hauptaufgabe der Klasse *AbstractComponentConfiguration* ist die Realisierung des Zustandsautomaten der Klasse *ComponentConfiguration* des Komponentenmodells. Zur Umsetzung wird, genau wie zuvor, das Entwurfsmuster *Zustand* verwendet. Die Schnittstelle *IStateCC* definiert hierzu sämtliche Trigger des Zustandsautomaten als Methoden. Das Verhalten der Komponentenkonfiguration innerhalb eines Zustandes wird durch die Klassen *StateCCResolved*, *StateCCResolving* sowie *StateCCNotResolved* realisiert.

Die einzige abstrakte Methode der Klasse *AbstractComponentConfiguration* ist *notifyStateChanged*. Diese muss vom Komponentenentwickler überschrieben werden und kann dazu verwendet werden, auf bestimmte Zustandsübergänge innerhalb der Anwendung zu reagieren.

7.1.5 ProvidedService

Die Klasse *ProvidedService* des Komponentenmodells wird im Framework durch die abstrakte Klasse *AbstractProvidedService* realisiert. Die Klassen und Schnittstellen, die zur Realisierung der im Komponentenmodell spezifizierten Funktionalität beitragen, sind in Abbildung 7-5 dargestellt.

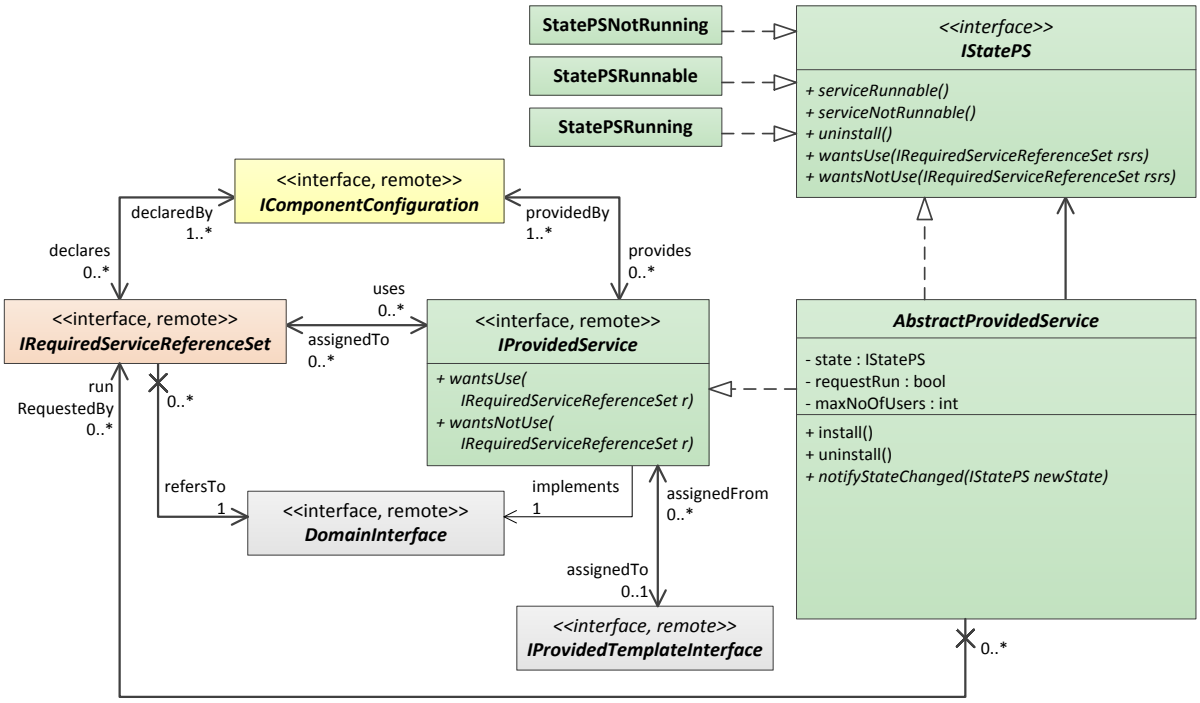


Abbildung 7-5: Realisierung der Klasse *ProvidedService* des Komponentenmodells

Der Zustandsautomat ist wie bei den Klassen *DynamicAdaptiveComponent* und *ComponentConfiguration* des Komponentenmodells auch unter Verwendung des Entwurfsmusters *Zustand* realisiert worden. Das Verhalten des *ProvidedService* im jeweiligen Zustand ist innerhalb der Klassen *StatePSNotRunning*, *StatePSRunnable* sowie *StatePSRunning* implementiert.

Die Klasse *AbstractProvidedService* implementiert des Weiteren die Schnittstelle *IProvidedService*. Über diese Schnittstelle können andere Komponenten über die Methoden *wantsUse* bzw. *wantsNotUse* einen Nutzungswunsch an den Dienst richten bzw. diesen widerrufen. Außerdem kann über die Methode *getImplements* die Domänenschnittstelle erfragt werden, welche durch den Dienst implementiert wird.

Die einzige abstrakte Methode der Klasse *AbstractProvidedService* ist *notifyStateChanged*. Diese muss vom Komponentenentwickler überschrieben werden und wird vom Framework immer dann aufgerufen, wenn ein Zustandswechsel innerhalb des *ProvidedService* stattfindet.

7.1.6 RequiredServiceReferenceSet

Die Klasse *RequiredServiceReferenceSet* des Komponentenmodells wird im Framework durch die Klasse *AbstractRequiredServiceReferenceSet* repräsentiert. Die zugehörigen Klassen und Schnittstellen des Frameworks sind in Abbildung 7-6 dargestellt.

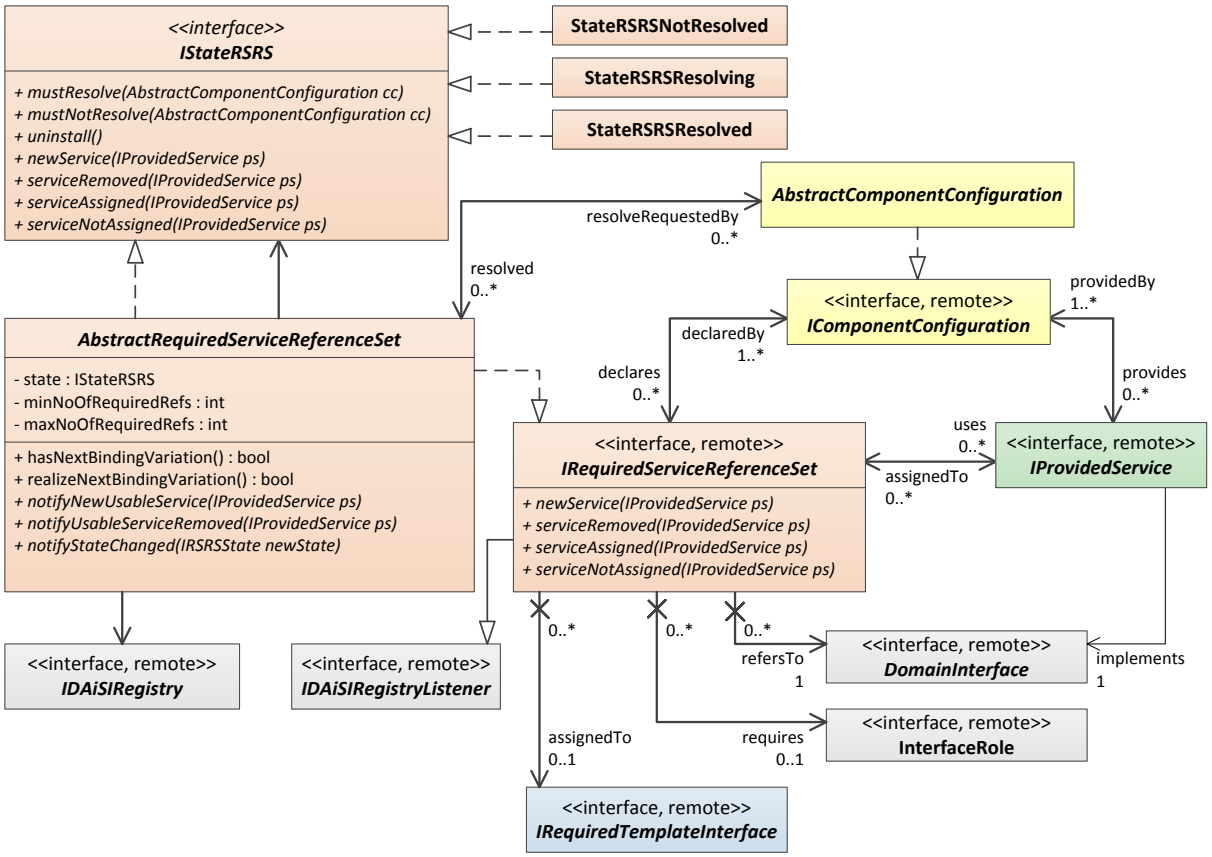


Abbildung 7-6: Realisierung von *RequiredServiceReferenceSet* des Komponentenmodells

Die Klasse *AbstractRequiredServiceReferenceSet* ist im Wesentlichen dafür verantwortlich, die Menge der verwendeten Dienste zu verwalten. Der entsprechende Zustandsautomat hierzu wird wieder über das Entwurfsmuster *Zustand* realisiert, welches mit Hilfe der Klassen *StateRSRSNotResolved*, *StateRSRSResolving* und *StateRSRSResolved* sowie der Schnittstelle *IStateRSRS* umgesetzt wurde. Um die Menge der zur Verfügung stehenden Dienste aktuell zu halten, implementiert die Klasse *AbstractRequiredServiceReferenceSet* die Schnittstelle *IDAiSIRegistryListener* und ist zudem als *Observer* bei der *Registry* registriert.

Über die Schnittstelle *IRequiredServiceReferenceSet* wird die Klasse *AbstractRequiredServiceReferenceSet* durch *ProvidedServices* anderer Komponenten darüber informiert, ob ihr der Dienst zugeteilt wurde bzw. wieder entzogen wurde. Hierzu dienen die Methoden *serviceAssigned* bzw. *serviceNotAssigned* der Schnittstelle.

Neben einer Implementierung der Schnittstellen *IDAiSIRegistryListener* und *IRequiredServiceReferenceSet* implementiert die Klasse *AbstractRequiredServiceReferenceSet* zwei weitere Methoden, nämlich *hasNextBindingVariation* und *realizeNextBindingRealization*. Diese dienen im Wesentlichen der Realisierung des Konfigurationsalgorithmus, welcher in Abschnitt Listing 6-2 vorgestellt wurde. Mit Hilfe dieser Methode kann während der Konfigurationsphase über die möglichen Belegungen der Menge *uses* iteriert und eine Belegung umgesetzt werden. Wird eine Komponente einem *Template* zugeordnet, so werden einzelne *RequiredServiceReferenceSets* der Komponente ggf. einem *RequiredTemplateInterface* zugeordnet. Diese Zuordnung wird über die Assoziation *assignedTo* abgebildet. Eine derartige Zuordnung hat Auswirkungen auf die möglichen Belegungen der Menge *uses*, wie bereits in Kapitel 6 erläutert wurde.

Die abstrakte Klasse *AbstractRequiredServiceReferenceSet* definiert drei abstrakte Methoden, welche durch den Komponentenentwickler überschrieben werden müssen. Diese Methoden werden später durch das Framework aufgerufen, um die Komponente zum einen über Veränderungen der Menge *uses* (*notifyNewUsableService* und *notifyUsableServiceRemoved*), sowie über eine Änderung des Ausführungszustandes (*notifyStateChanged*) zu informieren.

In den vorangegangenen Abschnitten wurde das Framework zur Entwicklung selbstadaptiver Komponenten skizziert. Auf eine detaillierte Beschreibung der Interaktion der einzelnen Klassen wurde hierbei bewusst verzichtet, da eine entsprechende Verhaltensspezifikation bereits in Form von Zustandsautomaten in den Kapiteln 4 und 5 vorgenommen wurde.

Im nun folgenden Abschnitt wird nun ein Überblick über das Framework zur Spezifikation von Anwendungsarchitekturen für selbstorganisierende Anwendungen gegeben.

7.1.7 Application

Die Klasse *Application* des zuvor vorgestellten Modells wird im Framework durch die abstrakte Klasse *AbstractApplication* umgesetzt. Diese Klasse sowie deren Beziehungen zu anderen Klassen und Schnittstellen des Frameworks ist in Abbildung 7-7 dargestellt.

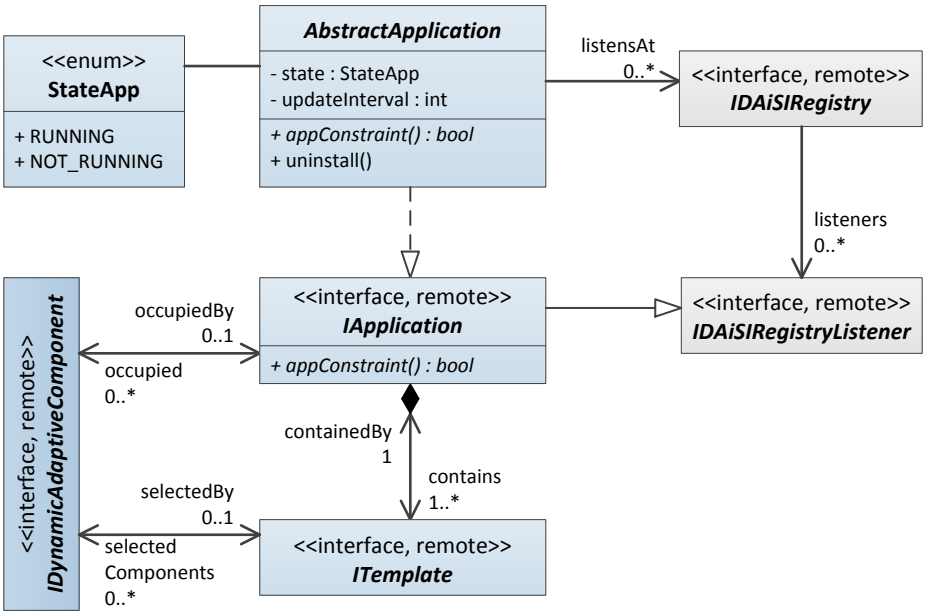


Abbildung 7-7: Realisierung der Klasse *Application* innerhalb des Frameworks

Die Klasse *AbstractApplication* ist verantwortlich für die Steuerung des Konfigurationsprozesses der Anwendung. Sie besitzt als einzige Methode die abstrakte Methode *appConstraint*, welche durch den Anwendungsentwickler überschrieben werden muss. Diese Methode realisiert das Prädikat *appConstraint*, welches bereits in Kapitel 6 vorgestellt wurde. Über die Assoziationen *contains* und *selectedComponents* können innerhalb der Methode die entsprechenden Anforderungen an die Menge spezifiziert werden. Liefert die Methode *true* zurück, so wurde eine gültige Anwendungskonfiguration gefunden, ansonsten muss eine neue Konfiguration erzeugt werden.

Die Klasse *AbstractApplication* benötigt stets Informationen über die zur Verfügung stehenden Komponenten, um auf dessen Basis eine Konfiguration zu erzeugen. Hierzu implementiert sie

die Schnittstelle *IDAiSIRegistryListener* und registriert sich bei dessen Instanziierung bei der *Registry* durch Aufruf von *registerListener*. Fortan wird die Anwendung über hinzukommende und wegfallende Komponenten informiert und kann entsprechend reagieren. So müssen beispielsweise bei Wegfall einer Komponente sämtliche Bedingungen der Anwendung neu überprüft werden und ggf. eine neue geeignete Konfiguration gesucht werden.

Nach Instanziierung der Anwendung werden zunächst Komponenten für die Anwendung durch Aufruf von *occupy* reserviert. Die Menge an erfolgreich reservierten Komponenten dient anschließend als Basis für den Konfigurationsprozess. Dann wird eine erste Zuordnung von Komponenten zu *Templates* vorgenommen. Hierzu werden beim jeweiligen *Template* die Methoden *addSelectedComponent* bzw. *removeSelectedComponent* verwendet (siehe folgende Abschnitte). Bei dieser Zuordnung wird bereits auf die Einhaltung der minimalen und maximalen Anzahl an erforderlichen Komponenten je *Template* geachtet. Innerhalb der Methode *addSelectedComponent* wird der betreffenden Komponente durch Aufruf von *selectedBy* mitgeteilt, welchem *Template* sie zugewiesen wurde.

Nach der Zuordnung einer Komponente zu einem *Template* kann diese die möglichen Variationen berechnen, die sich aus der Zuordnung ergeben. Hierzu zählen Variationen bzgl. der Zuordnung von *ProvidedServices* zu *ProvidedTemplateInterfaces* bzw. von *RequiredServiceReferenceSets* zu *RequiredTemplateInterfaces*, sowie mögliche Variationen bzgl. der Menge der zu verwendenden Dienste innerhalb von *RequiredServiceReferenceSets*. Die Komponente implementiert die Methoden *hasNextVariation* bzw. *realizeNextVariation*, um neue Anwendungskonfigurationen zu erzeugen (siehe Abschnitt 7.1.3).

Befindet sich eine Komponente im Zustand RUNNING, so wird zyklisch überprüft, ob die erzeugte Konfiguration noch konform zur Anwendungsarchitektur ist. Das Überprüfungsintervall kann mit Hilfe des Attributs *updateInterval* der Klasse *AbstractApplication* festgelegt werden.

7.1.8 Template

Die Klasse *Template* des Modells aus Kapitel 6 wird im Framework im Wesentlichen durch die Klasse *Template* umgesetzt. Durch ein *Template* werden, wie bereits zuvor beschrieben, die Variationsmöglichkeiten der Konfiguration einer Komponente beschränkt. So muss sie nach einer Zuordnung zu einem *Template* in *RequiredServiceReferenceSets* die Grenzen bzgl. der Anzahl zu verwendender Dienste einhalten, welche in den Attributen *minNoOfRequiredRefs* des jeweils zugeordneten *RequiredTemplateInterface* hinterlegt sind. Damit die Komponente hierzu in der Lage ist, benötigt sie Zugriff auf sämtliche Anforderungen, die durch das *Template* gegeben sind. In Abbildung 7-8 sind die erforderlichen Schnittstellen und Klassen im Zusammenhang dargestellt.

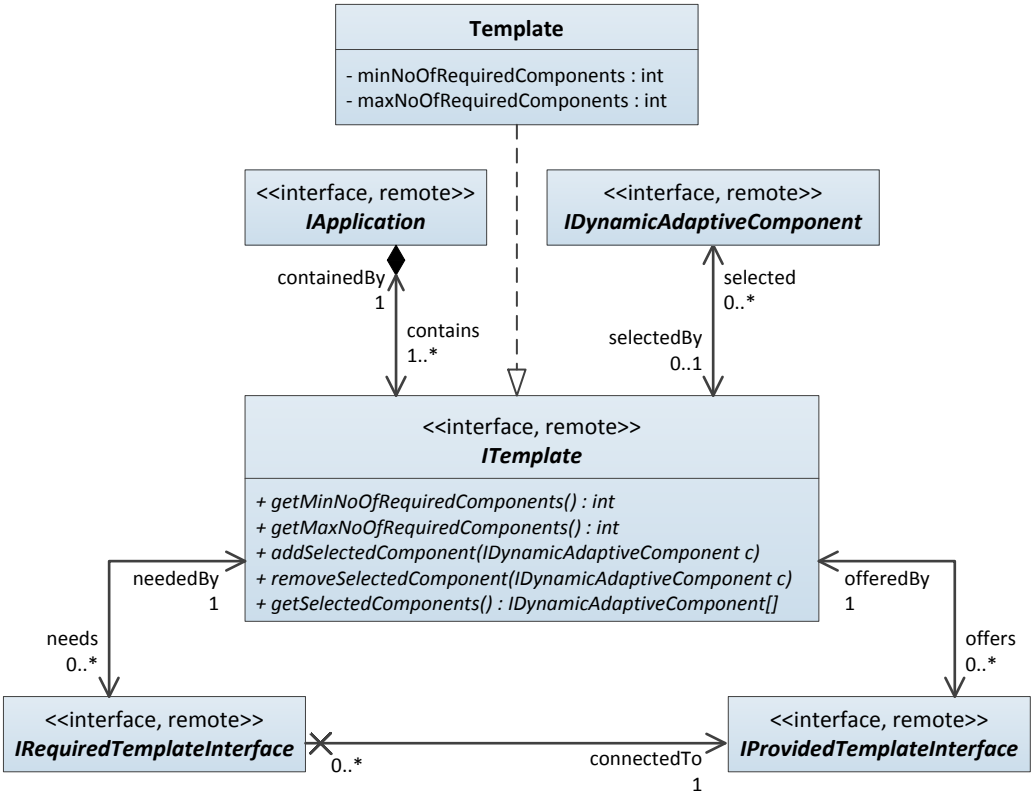


Abbildung 7-8: Die Realisierung der Klasse *Template* aus dem Modell

Zugriff auf die gegebenen Anforderungen bekommt die Komponente über die Assoziation *selectedBy*. Wird bei einer Komponente die Methode *selectedBy* durch die Anwendung (*Application*) aufgerufen (siehe Abschnitt 7.1.3), so wird das entsprechende Feld mit der übergebenen Referenz auf das entsprechende *Template* belegt. Über die Schnittstelle *ITemplate* kann die Komponente dann abfragen, welche *ProvidedTemplateInterfaces* und *RequiredTemplateInterfaces* das ihr zugeordnete *Template* definiert und entsprechend die möglichen Variationen berechnen.

Bei den Klassen *ProvidedTemplateInterface* und *RequiredTemplateInterface* handelt es sich um die Umsetzung der entsprechenden Klassen des Modells aus Kapitel 6. Über die Schnittstellen *IRequiredTemplateInterface* und *IProvidedTemplateInterface* können Komponenten auf die benötigten Informationen zugreifen. Die Zuordnung von *ProvidedServices* und *RequiredServiceReferenceSets* zu entsprechenden *IProvidedTemplateInterfaces* bzw. *IRequiredTemplateInterfaces* ist über die Assoziation *assignedTo* realisiert. Zur Bestimmung derjenigen Dienste, welche innerhalb eines *RequiredServiceReferenceSets* verwendet werden dürfen, muss die Komponente zunächst vom jeweiligen *RequiredServiceReferenceSet* zum zugeordneten *RequiredTemplateInterface* über die Assoziation navigieren. Mit Hilfe der Assoziation *connectedTo* erhält sie dann Zugriff auf das verbundene *ProvidedTemplateInterface*. Über die Schnittstelle *IProvidedTemplateInterface* können dann wiederum die zugeordneten *ProvidedServices* erfragt werden. Diese bilden dann die Menge der Dienste, die für das *RequiredServiceReferenceSet* für eine Verwendung in Betracht kommen.

7.1.9 ProvidedTemplateInterface und RequiredTemplateInterface

Die Klassen *ProvidedTemplateInterface* und *RequiredTemplateInterface* des Modells sind in der Implementierung durch jeweils eine Schnittstelle repräsentiert, die wiederum durch jeweils eine Klasse implementiert wird. Abbildung 7-9 zeigt die Schnittstellen und Klassen in Zusammenhang mit den assoziierten Klassen und Schnittstellen.

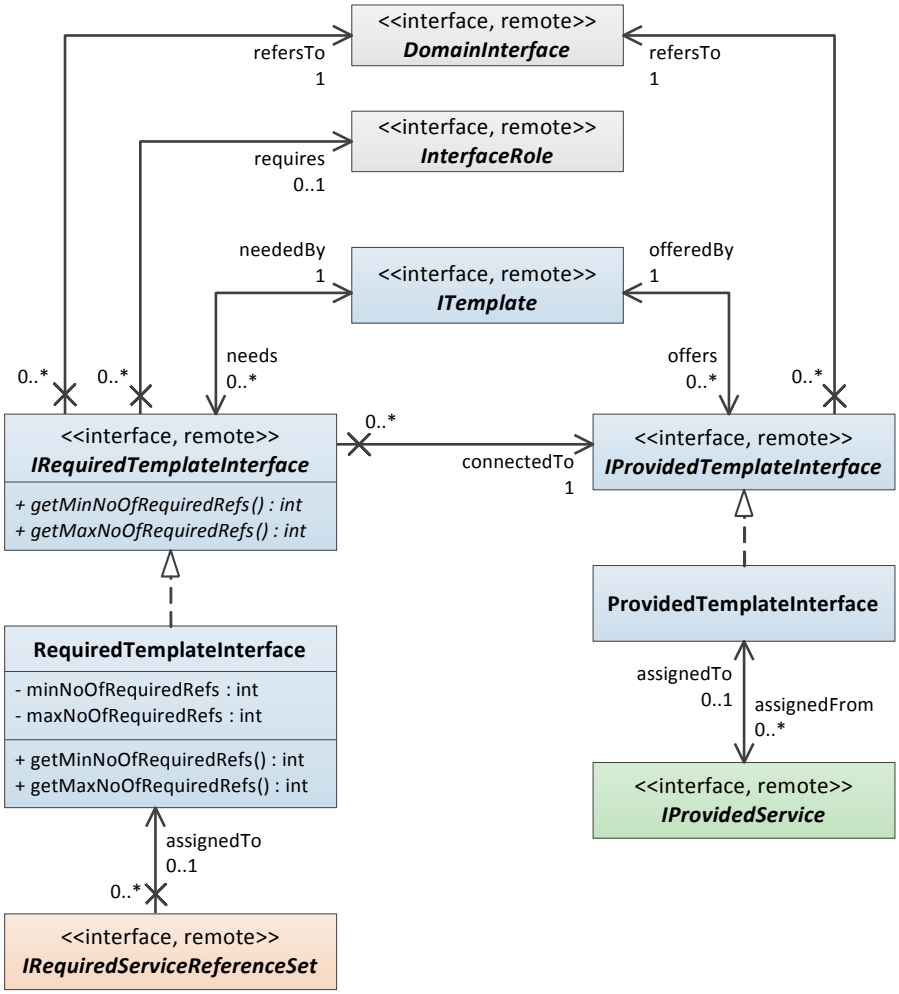


Abbildung 7-9: Realisierung der Klassen *ProvidedTemplateInterface* und *RequiredTemplateInterface* des Modells

Diese Klassen besitzen keinerlei Logik, sondern dienen in erster Linie der Spezifikation von Bedingungen an benötigte und angebotene Dienste. So kann über die Schnittstelle *IRequiredTemplateInterface* während des Konfigurationsprozesses das Intervall abgefragt werden, welches einem *RequiredServiceReferenceSet* später Vorgaben hinsichtlich der Anzahl benötigter Dienste macht.

Bei dem hier vorgestellten Framework handelt es sich um eine prototypische Umsetzung der vorgestellten Konzepte, die lediglich die Umsetzbarkeit des Lösungskonzeptes zeigt. Im folgenden Abschnitt wird abschließend skizziert, wie die Anwendungsentwicklung auf Basis des Frameworks umgesetzt werden kann.

7.2 Entwicklung und Ausführung einer Anwendung auf Basis des Frameworks

In diesem Abschnitt wird nun abschließend skizziert, wie das zuvor vorgestellte Framework zur Entwicklung einer dynamisch-adaptiven Anwendung verwendet werden kann, beginnend bei der Definition und Bereitstellung des Domänenmodells bis hin zur Ausführung der Anwendung. Als Beispielanwendung dient das bereits aus Kapitel 3.1 bekannte Beispiel eines Unterstützungssystems für Biathleten.

7.2.1 Definition eines Domänenmodells

Zu Beginn der Anwendungsentwicklung auf Basis des entwickelten Frameworks steht die Definition des Domänenmodells. Hierzu zählen insbesondere die Domänenschnittstellen sowie die Schnittstellenrollen, die für die Biathlondomäne benötigt werden. Eine der benötigten Domänenschnittstellen ist *IAthlete*, wie sie bereits in Abschnitt 3.1 angegeben wurde. Die Definition dieser Schnittstelle für die Infrastruktur ist in Listing 7-1 angegeben:

```
1  public interface IAthlete extends DomainInterface {
2    public String getName() throws RemoteException;
3    public String getTeamName() throws RemoteException;
4    public int getPulse() throws RemoteException;
5    public String getSkiingStyle() throws RemoteException;
6  }
```

Listing 7-1: Definition der Domänenschnittstelle *IAthlete*

Hierbei ist lediglich zu beachten, dass die Schnittstelle von *DomainInterface* des Frameworks erbt. Des Weiteren müssen sämtliche Methoden die Exception *RemoteException* werfen. Dies hängt ausschließlich mit der Verwendung von RMI zusammen.

Die Spezifikation einer Schnittstellenrolle geschieht durch Implementierung der Schnittstelle *InterfaceRole* des Frameworks (siehe Abschnitt 7.1.1). Hierzu müssen die Methoden *isConformTo* und *getInterfaceRoleName* implementiert werden. Listing 7-2 zeigt die Implementierung der Rolle *LeftStickRole*:

```
1  public class LeftStickRole
2              extends UnicastRemoteObject
3              implements InterfaceRole {
4
5    public LeftStickRole() throws RemoteException {
6      super();
7    }
8
9    @Override
10   public boolean isConformTo(IProvidedService ps) {
11     try {
12       return result = ((IStick) ps).getSide().equals("left");
13     } catch (RemoteException e) {...}
14   }
15
16   @Override
17   public String getInterfaceRoleName() {
18     return "LeftStickRole";
19   }
```

```
20 }
```

Listing 7-2: Implementierung der Schnittstellenrolle *LeftStickRole*

Bei der Implementierung einer Schnittstellenrolle sind einige Dinge zu beachten, die im Wesentlichen damit zusammenhängen, dass RMI zur Kommunikation verwendet wird. Zum einen muss jede Schnittstellenrolle von *UnicastRemoteObject* erben und die Schnittstelle *InterfaceRole* implementieren. Die Methode *isConformTo* liefert im konkreten Fall genau dann *true* zurück, wenn der als Parameter übergebene Dienst bei Aufruf der Methode *getSide* die Zeichenkette „left“ zurückliefert. Um diese Methode aufrufen zu können, muss der übergebene Dienst zunächst auf die passende Domänenschnittstelle gecastet werden (in diesem Fall auf *IStick*). Damit dieser Cast funktionieren kann, muss bei der Entwicklung der Komponenten darauf geachtet werden, dass einem *RequiredServiceReferenceSet* lediglich eine Schnittstellen-Rolle zugeordnet wird, die typkompatibel zur referenzierten Domänenschnittstelle ist (hierzu später mehr).

Auf diese Weise lassen sich nun sämtliche Domänenschnittstellen und Schnittstellenrollen entwickeln. Wie diese anschließend im System bekannt gemacht werden, wird später in Abschnitt 7.2.4 beschrieben.

7.2.2 Entwicklung von Komponenten

Ist das Domänenmodell einmal entwickelt worden, können darauf aufbauend die einzelnen Komponenten entwickelt werden. Ebenso kann unabhängig von den Komponenten die Anwendung spezifiziert werden (siehe nächster Abschnitt).

Als Beispiel sei angenommen, dass eine Komponente entwickelt werden soll, die den Namen, die Teamzugehörigkeit sowie den aktuellen Puls eines Sportlers anzeigt. Die Struktur der Komponente sowie eine mögliche Sicht für den Anwender ist in Abbildung 7-10 dargestellt.

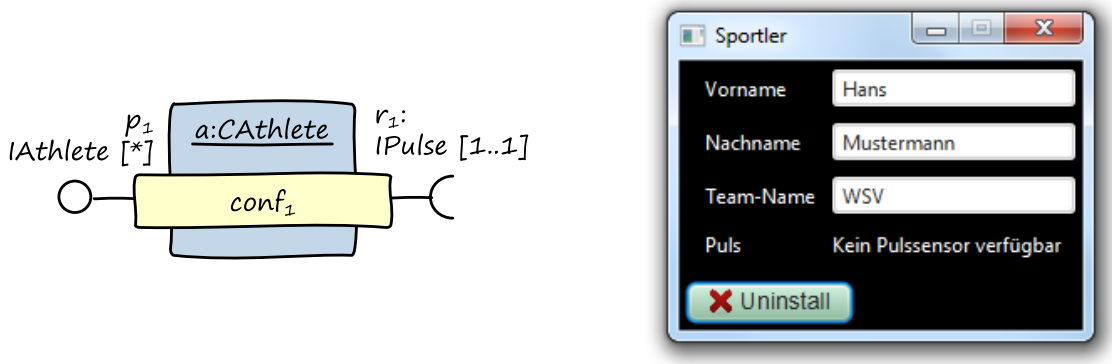


Abbildung 7-10: Beispiel einer einfachen Sportlerkomponente

In Abbildung 7-11 wird mittels eines UML Klassendiagramms gezeigt, wie diese Komponente mit Hilfe der Frameworkklassen entwickelt werden kann.

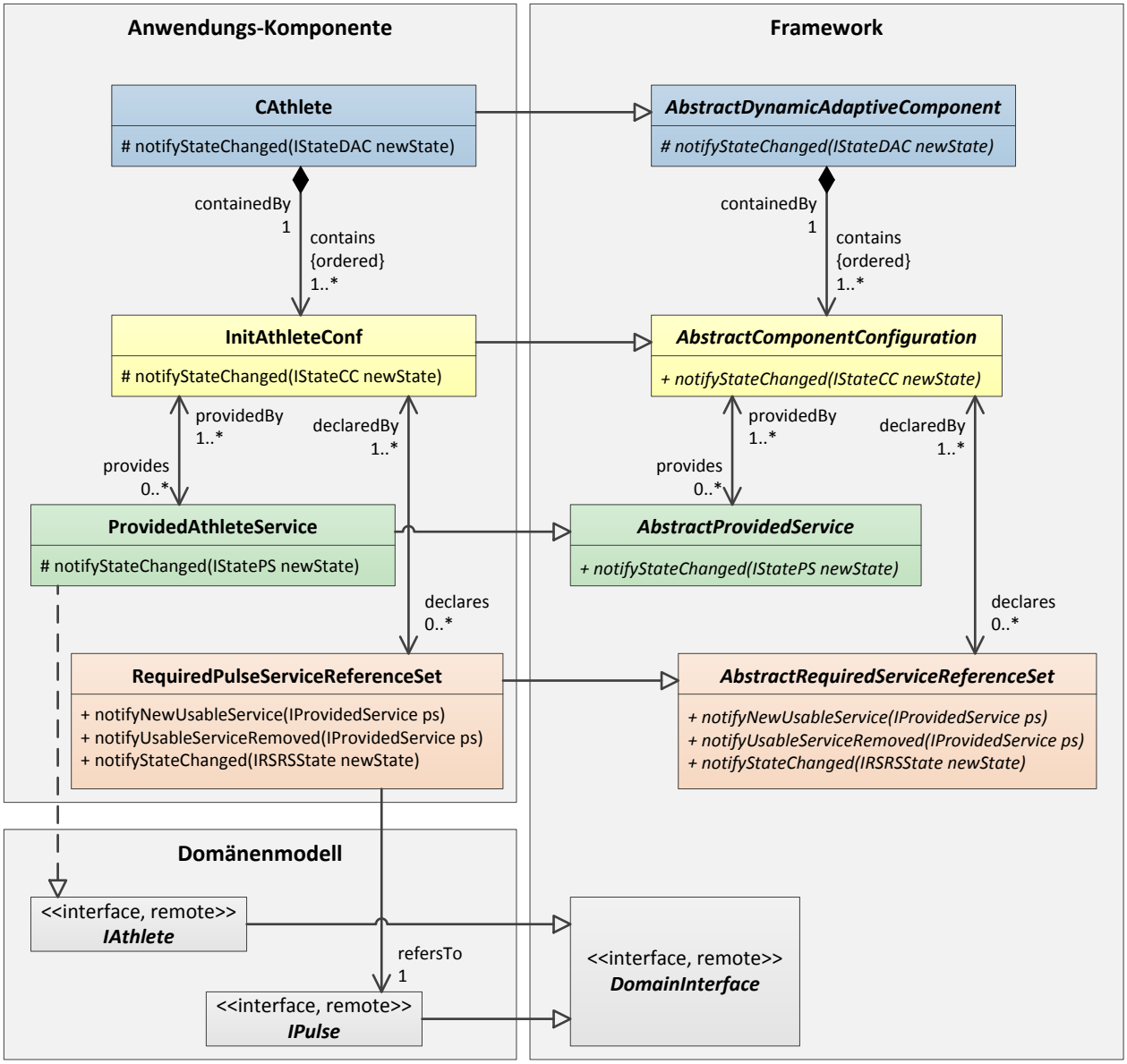


Abbildung 7-11: Beispiel zur Frameworknutzung

In Listing 7-3 und folgende sind die wesentlichen Auszüge aus einer prototypischen Implementierung namens *CAthlete* angegeben. Diese werden im Folgenden näher erläutert:

```
1  public class CAthlete extends AbstractDynamicAdaptiveComponent {
2
3    private AthleteGui athleteGui;
4    IPulse pulseSensor = null;
5
6    protected CAthlete() throws RemoteException {
7      super();
8    }
9
10   public static void main(String[] args) {
11     Application.launch(JavaFXAthleteApp.class, args);
12   }
13
14   void initComponentStructure(AthleteGui athleteGui) {
15     this.athleteGui = athleteGui;
```

```
16      try {
17        AbstractComponentConfiguration conf1 = new InitAthleteConf();
18        AbstractProvidedService p1 = new ProvidedAthleteService();
19        AbstractRequiredServiceReferenceSet r1 = new
20                                          RequiredPulseServiceReferenceSet();
21
22        conf1.declares.add(r1);
23        r1.declaredBy.add(conf1);
24
25        conf1.provides.add(p1);
26        p1.providedBy.add(conf1);
27
28        this.contains.add(conf1);
29        conf1.containedBy = this;
30
31      } catch (RemoteException e) {...}
32    }
33
34    @Override
35    protected void notifyStateChanged(IStateDAC newState) {...}
36
37    // Definition der folgenden privaten Klassen
38    // InitAthleteConf (siehe Listing 7-5)
39    // ProvidedAthleteService (siehe Listing 7-6)
40    // RequiredPulseServiceReferenceSet (siehe Listing 7-7)
41
42    ...
43  }
44
45
```

Listing 7-3: Auszüge aus einer prototypischen Implementierung einer Sportlerkomponente.

Die Implementierung einer Komponente besteht aus mindestens einer Klasse, welche von der abstrakten Frameworkklasse *AbstractDynamicAdaptiveComponent* erbt. Hierbei handelt es sich im Grunde um die Umsetzung der *DynamicAdaptiveComponent* aus dem Komponentenmodell.

In Zeile 4 ist für diese Komponente eine Variable namens *pulseSensor* vom Typ *IPulse* deklariert. In diese wird später eine Referenz auf einen *ProvidedService* einer anderen Komponente hinterlegt, der die Schnittstelle *IPulse* implementiert.

In den Zeilen 6-8 ist der Konstruktor angegeben. Hier ist es wichtig, dass innerhalb des Konstruktors der Konstruktor der Oberklasse (*AbstractDynamicAdaptiveComponent*) durch Aufruf von *super()* aufgerufen wird. Hierdurch werden innerhalb des Frameworks erste Initialisierungen vorgenommen.

Die grafische Oberfläche der hier dargestellten Komponente wurde mit Hilfe von JavaFX [WGC+11] erzeugt. Der Code innerhalb der *main*-Methoden (Zeilen 10-12) sind ausschließlich JavaFX-spezifisch und sorgen dafür, dass die Methode *start* der Klasse *JavaFXAthleteApp* ausgeführt wird (siehe Listing 7-4):

```
1  public class JavaFXAthleteApp extends Application {
2
3    @Override
4    public void start(Stage stage) throws Exception {
5      CAthlete cAthlete = new CAthlete();
6      AthleteGui athleteGui =
7          new AthleteGui(stage, cAthlete, "Hans", "Mustermann", "WSV");
8      cAthlete.initComponentStructure(athleteGui);
9    }
10 }
```

Listing 7-4: Die JavaFX-Anwendungsklasse

Innerhalb dieser Klasse wird zunächst eine Instanz der Komponente erzeugt, dann eine Instanz der GUI-Realisierung und anschließend durch Aufruf der Methode *initComponentStructure* die Komponentenstruktur angelegt. Diese Methode ist in Listing 7-3 in den Zeilen 14 bis 32 angegeben und wird nun detailliert beschrieben.

In den Zeilen 17 bis 20 wird zunächst Instanzen von einer *ComponentConfiguration* erzeugt, welche durch die Klasse *InitAthleteConf* umgesetzt wurde (siehe Listing 7-5). Außerdem wird die Instanz eines *ProvidedServices* und eines *RequiredServiceReferenceSets* erzeugt, welche mittels der Klassen *ProvidedAthleteService* (Listing 7-6) und *RequiredPulseServiceReferenceSet* (Listing 7-7) implementiert wurden. In den Zeilen 22 bis 29 werden diese einzelnen Elemente anschließend miteinander verknüpft.

So wird in den Zeilen 22 und 23 das *RequiredServiceReferenceSet* mit der *ComponentConfiguration* verknüpft. In den Zeilen 25 und 26 wird selbiges mit dem *ProvidedService* durchgeführt. Und abschließend wird die *ComponentConfiguration* mit der Komponente verknüpft (Zeilen 28 und 29). Es ist hierbei insbesondere darauf zu achten, dass die entsprechende Verknüpfung stets in beide Richtungen angelegt wird, also beispielsweise vom *ProvidedService* zur *ComponentConfiguration* und umgekehrt. Nur so kann das Framework später für eine korrekte Konfiguration der Komponente sorgen.

Abschließend muss die Komponente noch die einzige abstrakte Methode der Oberklasse *AbstractDynamicAdaptiveComponent* implementieren, nämlich *notifyStateChanged*.

Die Implementierung der einzelnen Komponentenelemente (*ComponentConfiguration, ProvidedService* und *RequiredServiceReferenceSet*) wurde mittels privater Klassen innerhalb der Klasse *CAthlete* realisiert. Diese Klassen werden im Folgenden kurz erläutert.

Zur Implementierung einer *ComponentConfiguration* muss die realisierende Klasse lediglich von der abstrakten Frameworkklasse *AbstractComponentConfiguration* erben und die abstrakte Methode *notifyStateChanged* überschreiben (siehe Listing 7-5):

```
1  class InitAthleteConf extends AbstractComponentConfiguration {
2    protected InitAthleteConf() throws RemoteException {
3      super();
4    }
5
6    @Override
7    protected void notifyStateChanged(IStateCC newState) {...}
8  }
```

Listing 7-5: Realisierung einer Komponentenkonfiguration für die Athletenkomponente

Genau wie zuvor muss auch hier der Konstruktor der Oberklasse mittels *super* aufgerufen werden, damit das Framework die notwendigen internen Initialisierungen vornehmen kann.

Der *ProvidedService* der Komponente ist mittels der Klasse *ProvidedAthleteService* implementiert worden. Jeder *ProvidedService* muss zum einen von der abstrakten Frameworkklasse *AbstractProvidedService* erben, und zum anderen eine Domänenschnittstelle implementieren (siehe Listing 7-6):

```
1  class ProvidedAthleteService extends AbstractProvidedService
2                               implements IAthlete {
3
4    public ProvidedAthleteService() throws RemoteException {
5      super(IAthlete.class, true, -1);
6    }
7
8    @Override
9    protected void notifyStateChanged(IStatePS newState) {
10     System.out.println("State of ProvidedService p1: " + newState);
11     switch (newState) {
12       case RUNNING:
13         athleteGui.enableAthleteService(pulseSensor);
14         break;
15       case NOT_RUNNING:
16       case RUNNABLE:
17       case FINAL:
18         athleteGui.disableAthleteService();
19         break;
20     }
21   }
22
23   @Override
24   public String getName() throws RemoteException {
25     return athleteGui.getFirstName() + " " + athleteGui.getLastName();
26   }
27
28   ...
29
30 }
```

Listing 7-6: Realisierung eines *ProvidedService* für die Athletenkomponente

Der Konstruktor der Oberklasse erfordert drei Parameter, nämlich die Schnittstelle, welche durch den Dienst implementiert wird, eine Information darüber, ob das Flag *requestRun* gesetzt werden soll oder nicht, sowie die maximale Anzahl der zulässigen gleichzeitigen Nutzer des Dienstes. Der Rest der Klasse besteht zum einen aus der Implementierung der abstrakten Methode *notifyStateChanged* sowie aus der Implementierung der Domänenschnittstelle (in diesem Fall *IAthlete*.

Letztes Element der Komponente ist ein *RequiredServiceReferenceSet*, dessen Implementierung in der Klasse *RequiredPulseServiceReferenceSet* angegeben ist (siehe Listing 7-7):

```
1  class RequiredPulseServiceReferenceSet
2                           extends AbstractRequiredServiceReferenceSet {
3
4    public RequiredPulseServiceReferenceSet() throws RemoteException {
5      super(IPulse.class, 1, 1);
6    }
7
8    @Override
9    protected void notifyStateChanged(IStateRSRS newState) {...}
10
11   @Override
12   protected void notifyNewUsableService(IProvidedService ps) {
13     pulseSensor = (IPulse) ps;
14   }
15
16   @Override
17   protected void notifyUsableServiceRemoved(IProvidedService ps) {
18     pulseSensor = null;
19   }
20 }
```

Listing 7-7: Realisierung eines *RequiredServiceReferenceSets* für die Athletenkomponente

Jedes *RequiredServiceReferenceSet* muss von der abstrakten Frameworkklasse *AbstractRequiredServiceReferenceSet* erben und dessen Konstruktor aufrufen. Dieser erwartet drei Parameter, nämlich diejenige Schnittstelle, für die eine Implementierung durch einen *ProvidedService* benötigt wird sowie die minimale und maximale Anzahl benötigter Dienste.

Zudem müssen drei abstrakte Methoden überschrieben werden, nämlich zum einen die Methode *notifyStateChanged* und zum anderen zwei Methoden, welche vom Framework immer dann aufgerufen werden, wenn ein neuer Dienst zur Nutzung bereitsteht oder ein bereits genutzter wegfällt. In diesem Fall wird bei Bereitstellung eines neuen Dienstes durch das Framework die entsprechende Referenz in der Variablen *pulseSensor* abgelegt (siehe Listing 7-3, Zeile 4). Somit kann innerhalb der Komponente auf diesen Dienst zugegriffen werden, um beispielsweise den aktuellen Puls abzufragen.

Auf diese Weise lassen sich nun beliebige Komponenten entwickeln. Um die Komponente auszuführen, muss lediglich die Komponente instanziiert werden und anschließend die Methode *install* der abstrakten Oberklasse *AbstractDynamicAdaptiveComponent* aufgerufen werden. Allerdings muss hierfür zuvor die *Registry* gestartet worden sein. Details zur Ausführung von Komponenten und Anwendungen werden später im Abschnitt 7.2.4 vorgestellt.

7.2.3 Entwicklung einer Anwendung

Neben dem Framework zur Entwicklung und Ausführung von Komponenten wurde im Rahmen der Arbeit ebenfalls ein Framework zur Spezifikation und Ausführung von Anwendungen entwickelt. Anhand eines kleinen Beispiels wird in diesem Abschnitt gezeigt, wie dieses Framework verwendet wird.

Als Beispiel dient ein kleiner Ausschnitt aus der bereits vorgestellten Biathlonanwendung. Das Beispiel besteht aus lediglich zwei miteinander verbundenen *Templates*, eines für Athletenkomponenten und eines für Pulskomponenten. Die grafische Repräsentation ist in Abbildung 7-12 dargestellt:

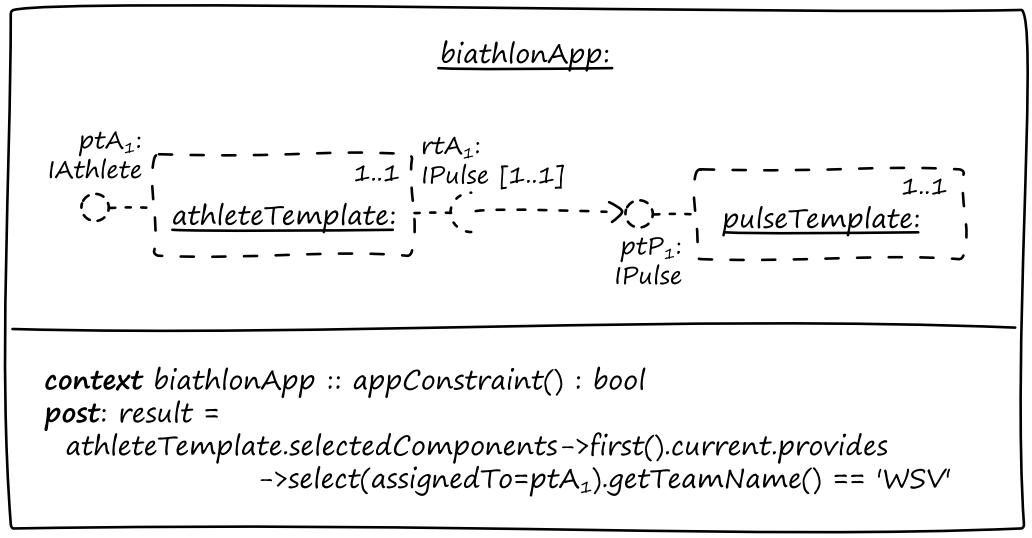


Abbildung 7-12: Kleines Anwendungsbeispiel

Eine mögliche Umsetzung dieser Spezifikation unter Verwendung des Frameworks ist in Listing 7-8 angegeben:

```
1  public class BiathlonApplication extends AbstractApplication {
2
3    public BiathlonApplication() {
4      try {
5        Template pulseTemplate = new Template(1, 1);
6        Template athleteTemplate = new Template(1, 1);
7
8        IProvidedTemplateInterface ptA1 = new
9                                ProvidedTemplateInterface(IAthlete.class);
10       IProvidedTemplateInterface ptP1 = new
11                               ProvidedTemplateInterface(IPulse.class);
12       IRequiredTemplateInterface rtA1 = new
13                               RequiredTemplateInterface(IPulse.class, 1, 1);
14
15       pulseTemplate.addOffers(ptP1);
16       ptP1.setOfferedBy(pulseTemplate)
17
18       athleteTemplate.addOffers(ptA1);
19       ptA1.setOfferedBy(athleteTemplate);
20
21       athleteTemplate.addNeeds(rtA1);
22       rtA1.setNeededBy(athleteTemplate);
23
24       rtA1.setConnectedTo(ptP1);
25
26       super.contains.add(athleteTemplate);
27       athleteTemplate.setContainedBy(this);
28
29       super.contains.add(pulseTemplate);
30       pulseTemplate.setContainedBy(this);
31     } catch (RemoteException e) {...}
32   }
33
34   @Override
35   protected boolean appConstraint() {
36     return (IAthlete)(athleteTemplate.getSelectedComponents().
```

```
37                          get(0).getCurrent().getProvides(ptA1)).
38                          getTeamName().equals("WSV");
39    }
40 }
```

Listing 7-8: Spezifikation einer Biathlonanwendung

Die Anwendung wurde mittels einer Klasse namens *BiathlonApplication* realisiert. Jede Anwendung muss von der abstrakten Frameworkklasse *AbstractApplication* erben. Innerhalb des Konstruktors kann die Anwendungsspezifikation umgesetzt werden unter Zuhilfenahme der Frameworkklassen und Methoden. In den Zeilen 5 und 6 werden die beiden *Template*-Objekte angelegt, wobei dem Konstruktor die minimale und maximale Anzahl benötigter Komponenten übergeben wird. In den Zeilen 8 bis 13 werden die *ProvidedTemplateInterfaces* und *RequiredTemplateInterfaces* erzeugt und in den Zeilen 15 bis 30 werden die erzeugten Elemente miteinander in Beziehung gesetzt.

Abschließend muss die abstrakte Methode *appConstraint* implementiert werden, welche in der abstrakten Oberklasse *AbstractApplication* deklariert ist. Innerhalb dieser Methode können Bedingungen formuliert werden, welche für Komponenten der Anwendung sowie deren Beziehungen untereinander gelten sollen. Der Einstiegspunkt hierzu bildet die Menge *selectedComponents* der einzelnen *Templates*, auf die mittels der Methode *getSelectedComponents* zugegriffen werden kann. Im Beispiel liefert die Methode *appConstraint* nur dann *true* zurück, wenn die gewählte Komponente des Athleten-*Templates* bei Aufruf von *getTeamName* den Wert *WSV* zurückliefert. Der Methode *getProvides* kann hierbei ein *ProvidedTemplateInterface* als Parameter übergeben werden. Es wird dann von dieser Methode derjenige *ProvidedService* zurückgegeben, welcher dem übergebenen *ProvidedTemplateInterface* zugeordnet ist.

Im folgenden Abschnitt wird abschließend erläutert, wie die Komponenten der Anwendung sowie die Anwendung selbst installiert und gestartet werden kann.

7.2.4 Installation und Ausführung einer Anwendung

Bevor Komponenten und Anwendungen gestartet werden können, müssen bestimmte Infrastrukturdienste gestartet werden. Anschließend müssen die benötigten Schnittstellenrollen bei der *Registry* registriert werden. Im nächsten Schritt können dann Komponenten und Anwendungen gestartet werden.

Die einzelnen Bestandteile der Infrastruktur können auf unterschiedlichen Rechnern installiert werden. Eine beispielhafte Verteilung der einzelnen Einheiten ist in Abbildung 7-13 dargestellt:

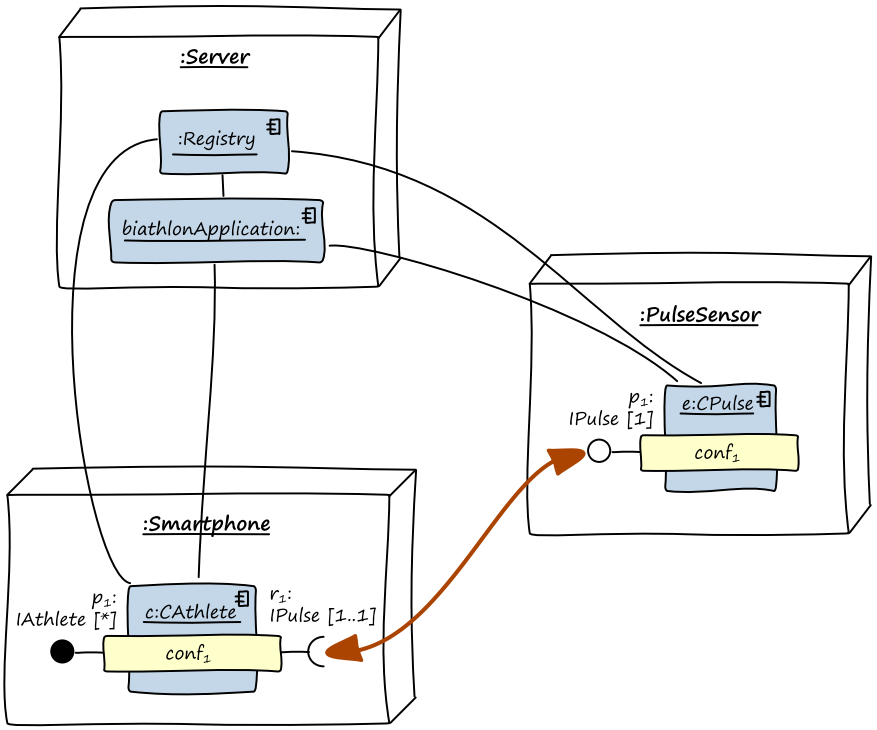


Abbildung 7-13: Beispiel für ein Deployment

Hierbei wird angenommen, dass *Registry*, die Komponenten und auch die Anwendung auf unterschiedlichen Rechnern installiert und ausgeführt werden. Die gesamte Infrastruktur ist in Form einer *jar*-Datei verfügbar, welche auf jedem Knoten vorhanden sein muss.

Das Starten der *Registry* geschieht durch Ausführen der Klasse *DAiSIRegistry* des Frameworks. Diese enthält eine *main*-Methode, welche alle weiteren Aufgaben übernimmt. Auch die einzelnen Komponenten, wie z.B. die zuvor vorgestellte Komponente *CAthlete*, kann durch einfaches Ausführen der *main*-Methode gestartet werden. Registriert wird die Komponente beim Framework jedoch erst bei Aufruf der Methode *install* der abstrakten Komponentenoberklasse *AbstractDynamicAdaptiveComponent*. Nach Aufruf der Methode sind die Komponente sowie dessen angebotene Dienste für andere Komponenten sichtbar und potentiell nutzbar. Durch Aufruf von *uninstall* kann die Komponente wieder aus der Infrastruktur entfernt werden.

8 Evaluierung

In diesem Kapitel wird die vorgestellte Lösung hinsichtlich ihrer Praxistauglichkeit sowie ihres Funktionalitätsumfangs evaluiert. Hierzu wird eine szenariobasierte Evaluierung vorgenommen [KBW+94, KKB+98]. Eine der wichtigsten Schritte bei szenariobasierten Verfahren ist das Erheben und Priorisieren von Szenarios. Jedes Szenario beschreibt hierbei eine typische Situation und typische Ereignisse, die auftreten können, sowie das gewünschte Verhalten. Im Rahmen der Evaluierung wird beschrieben, wie die Lösung dieses Szenario umsetzt. Die in dieser Arbeit vorgestellte Lösung wird anhand von fünf Szenarios evaluiert. Hierbei werden für jedes Szenario zunächst die Anforderungen an ein Framework für mobile, verteilte Anwendungen zusammengefasst. Anschließend wird anhand eines konkreten Beispiels gezeigt, wie das Framework diese Anforderungen umsetzt, sowie eine kurze Bewertung geliefert.

8.1 Szenario 1 – Automatische Ausführung anwendungsrelevanter Dienste

Ziel des Szenarios ist die Evaluation des Lösungskonzeptes hinsichtlich der Möglichkeit, das Startverhalten einer Anwendung basierend auf selbstorganisierenden Komponenten durch den Komponentenentwickler zu beeinflussen.

8.1.1 Anforderungen

Im einfachsten Fall besteht ein selbstorganisierendes System ausschließlich aus selbstorganisierenden Komponenten, welche gar keine, oder nur optionale Abhängigkeiten zu Diensten anderer Komponenten aufweisen. Typische Beispiele hierfür sind einfache Sensorkomponente, die lediglich Daten über angebotene Dienste anderen zur Verfügung stellen, wie zum Beispiel GPS-Dienste, die die aktuelle Position liefern, oder aber auch Komponenten, die Sensorwerte wie Temperatur, Lautstärke, Gewicht u.Ä. anbieten. Doch gerade diese Komponenten bieten einem Endanwender meist erst dann einen Nutzen, wenn sie in irgendeiner Art und Weise durch das System weiterverarbeitet werden. So ergeben die bereitgestellten Koordinaten eines GPS-Sensors für den Endanwender meist erst dann einen Nutzen, wenn die Position auf einer Karte angezeigt wird.

Auf der anderen Seite gibt es Komponenten, die bestimmte Funktionalitäten und Dienste bereitstellen, welche auch dann für einen Endanwender von Nutzen sind, wenn sie nicht durch andere Komponenten genutzt werden. Hierzu gehören insbesondere diejenigen Komponenten, welche eine Nutzungsschnittstelle für Endanwender realisieren. Dies können Komponenten sein, die eine Karte darstellen, den Temperaturverlauf der vergangenen Tage, die aktuelle Geschwindigkeit usw.

Gerade in mobilen verteilten Systemen werden derartige Komponenten häufig auf ressourcenschwachen Geräten ausgeführt, welche zudem meist mit einem Akku betrieben werden. Als Beispiel sei hier die GPS-Komponente eines Smartphones genannt. Gerade in dieser Art von Systemen steht die Einsparung von Ressourcen im Vordergrund, und Frameworks für derartige Systeme müssen diesen Aspekt berücksichtigen.

Die Lösung, welche in dieser Arbeit vorgestellt wurde, berücksichtigt diese Aspekte, indem zwei Arten von angebotenen Diensten unterschieden werden. Nämlich diejenigen, welche aus Sicht

des Komponentenentwicklers einen unmittelbaren Nutzen für den Endanwender darstellen (z.B. Komponenten, welche eine grafische Oberfläche realisieren), und diejenigen, die insbesondere zur Nutzung durch andere Komponenten entwickelt wurden (Sensorkomponenten). Nun ergibt es für die zweite Kategorie von Diensten keinen Sinn, diese auszuführen, wenn kein Nutzer in Form mindestens einer anderen Komponente vorhanden ist. Hier können Ressourcen geschont werden, wenn derartige Dienste tatsächlich erst für den Fall gestartet werden, wenn eine andere Komponente auf diesen Dienst angewiesen ist.

8.1.2 Umsetzung und Bewertung

Im vorgestellten Framework kann der Komponentenentwickler festlegen, um welche Art von Diensten eine Komponente anbietet. Hierzu muss er lediglich das Flag *requestRun* des entsprechenden Dienstes auf *true* setzen, falls es sich um einen Dienst handelt, der für Endanwender einen unmittelbaren Nutzen darstellt. Ansonsten wird das Flag auf *false* gesetzt. Das Framework sorgt zur Laufzeit dafür, dass die angebotenen Dienste der selbstorganisierenden Komponente nur dann ausgeführt werden, wenn sie auch tatsächlich benötigt werden.

Anhand zweier Beispiele wird die Arbeitsweise des Frameworks nun kurz erläutert. Hierfür sei zunächst angenommen, dass eine typische Sensorkomponente im System installiert wird, wie sie in Abbildung 8-1 dargestellt ist.

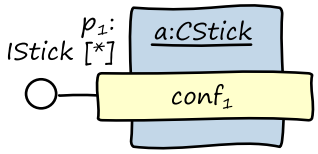


Abbildung 8-1: Beispiel einer Sensorkomponente

Das Flag *requestRun* wurde für den angebotenen Dienst nicht gesetzt, symbolisiert durch den nicht ausgefüllten Kreis. Der Dienst soll also erst ausgeführt werden, wenn ein Nutzer in Form einer anderen Komponente existiert. Das Framework prüft bereits bei der Installation der Komponente, ob das Flag gesetzt wurde oder nicht (siehe Abbildung 4-13 bzw. Listing 4-4). Ist das Flag nicht gesetzt, so verbleibt der Dienst im Zustand NOT_RUNNING, und verlässt ihn erst dann, wenn der Trigger *serviceRunnable* aufgerufen wird. Dieser Trigger teilt dem Dienst mit, dass er mit einer aktiven *ComponentConfiguration* verbunden ist. Liegen für den Dienst Nutzungsanfragen in Form anderer Komponenten vor, so wechselt der Dienst in den Zustand RUNNING, und ansonsten in den Zustand RUNNABLE. Der Automat für *ProvidedServices* stellt somit sicher, dass solche Dienste erst dann ausgeführt werden, wenn Nutzungsanfragen vorliegen. Im Beispiel liegen allerdings weder Nutzungsanfragen vor, noch ist der Dienst mit einer aktiven *ComponentConfiguration* verbunden, so dass sich am Zustand der Zustandsautomaten der Komponente nichts verändert.

Anders verhält es sich bei Komponenten, welche Dienste mit gesetztem *requestRun*-Flag anbieten. Ein Beispiel hierfür ist die bereits im Rahmen der Arbeit häufiger beschriebene Trainerkomponente, welche u.a. auch eine grafische Nutzungsschnittstelle für den Trainer realisiert. Für den angebotenen Dienst wurde das Flag gesetzt, was in diesem Fall eine ganze Reihe von Zustandsübergängen auslöst, welche in Abbildung 8-2 dargestellt sind und im Folgenden kurz erläutert werden.

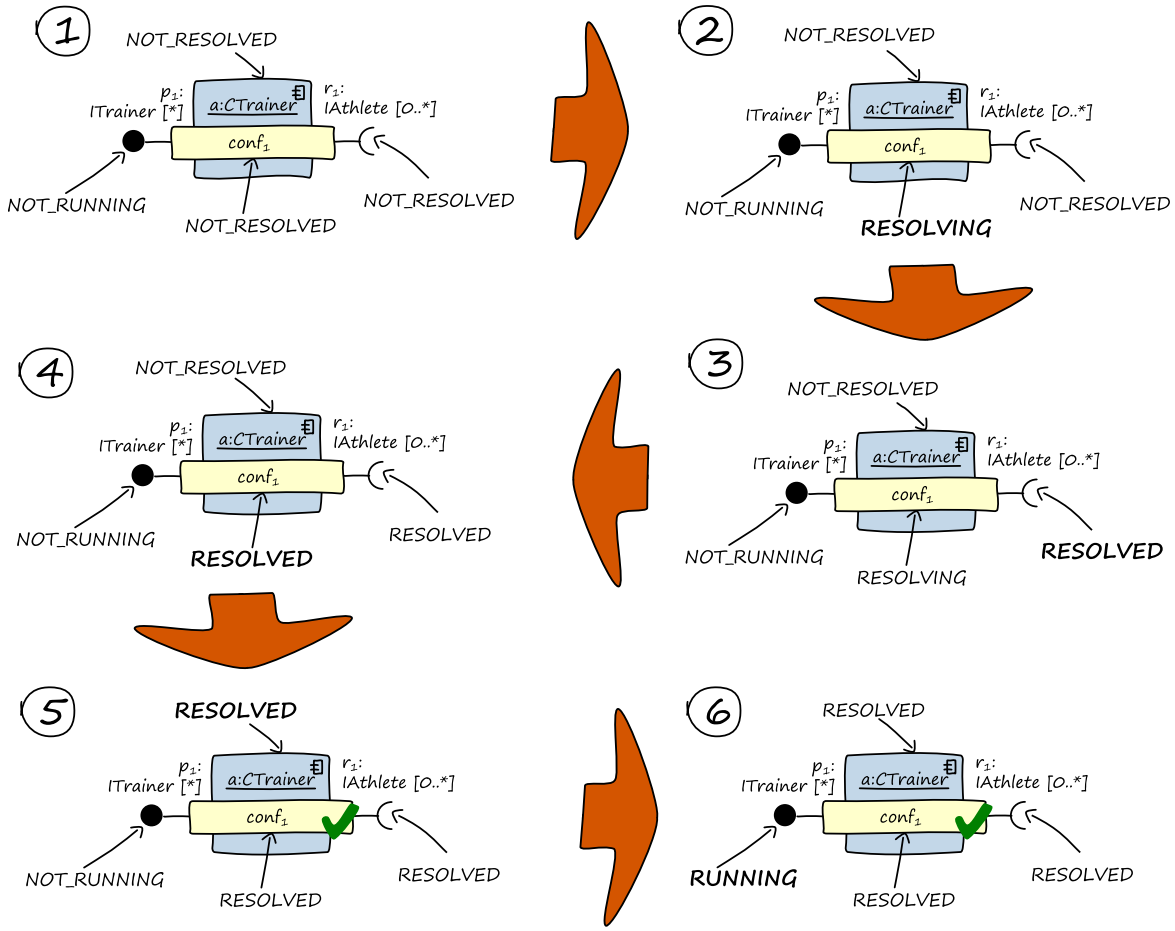


Abbildung 8-2: Beispiel zu gesetztem *requestRun*-Flag

Ausgangspunkt für den Rekonfigurationsprozess ist der angebotene Dienst, für den das Flag *requestRun* gesetzt ist (1). Der Automat aus Abbildung 4-13 prüft beim Eintritt in den initialen Zustand, ob das Flag gesetzt ist und ruft ggf. den Trigger *mustRun* bei allen *ComponentConfigurations* der Komponente auf. Diese beginnen anschließend mit der Auflösung von Abhängigkeiten zu anderen Komponenten, falls welche bestehen. Definiert eine *ComponentConfiguration* kein *RequiredServiceReferenceSet*, so wird der Komponente unmittelbar durch Aufruf des Triggers *configurationResolved* mitgeteilt, dass sie durch die Komponente aktiviert werden kann. Außerdem wechselt die *ComponentConfiguration* in den Zustand *RESOLVED*. Im Beispiel definiert die Konfiguration *config1* allerdings ein *RequiredServiceReferenceSet*. Dieses wird durch Aufruf des Triggers *mustResolve* (siehe Zustandsautomat in Abbildung 4-11) darüber informiert, dass es mit der Auflösung der Abhängigkeiten beginnen soll. Gleichzeitig wechselt der Automat der *ComponentConfiguration* in den Zustand RESOLVING und wartet dort auf die Nachricht des *RequiredServiceReferenceSets*, dass die Abhängigkeiten aufgelöst wurden (2).

Eine *ComponentConfiguration* verbleibt solange im Zustand RESOLVING, bis alle *RequiredServiceReferenceSets* der Konfiguration zurückgemeldet haben, dass deren Abhängigkeiten aufgelöst werden konnten. Nachdem dem *RequiredServiceReferenceSet* durch Aufruf von *mustResolve* mitgeteilt wurde, dass es mit der Auflösung von Abhängigkeiten beginnen soll, wechselt es entweder in den Zustand RESOLVING oder direkt in den Zustand RESOLVED, abhängig davon, welcher Wert im Attribut *minNoOfRequiredRefs* des

RequiredServiceReferenceSets hinterlegt ist. Ist der Wert 0 hinterlegt, gelten die Abhängigkeiten als aufgelöst und der Wechsel in den Zustand RESOLVED kann erfolgen, bei gleichzeitiger Benachrichtigung der verbundenen *ComponentConfigurations*. Andernfalls muss das *RequiredServiceReferenceSets* zunächst geeignete Dienste finden und für eine Verwendung reservieren. In diesem Fall findet zunächst ein Wechsel in den Zustand RESOLVING statt. Im Beispiel kann das *RequiredServiceReferenceSet* direkt in den Zustand RESOLVED wechseln und die *ComponentConfiguration* durch Aufruf von *rsrsResolved* darüber informieren (3).

Sobald eine *ComponentConfiguration* von allen verbundenen *RequiredServiceReferenceSets* die Meldung bekommen hat, dass die Abhängigkeiten zu Diensten anderer Komponenten aufgelöst werden konnten, kann auch die *ComponentConfiguration* in den Zustand RESOLVED wechseln und die Komponente hierüber durch Aufruf von *configurationResolved* informieren (4).

Eine Komponente wechselt immer dann in den Zustand RESOLVED, wenn sie mindestens eine *ComponentConfiguration* definiert, welche sich im Zustand RESOLVED befindet. Die *DynamicAdaptiveComponent* ist dafür verantwortlich, stets die bestmögliche *ComponentConfiguration* zu aktivieren sowie die verbundenen Dienste durch Aufruf von *serviceRunnable* darüber zu informieren, dass eine Ausführung des Dienstes nun möglich ist (5).

Im letzten Schritt entscheidet nun jeder benachrichtigte Dienst darüber, ob er ausgeführt werden muss oder nicht. Es gibt hier zwei Gründe, einen Dienst zu starten, nämlich die Existenz eines potentiellen Nutzers in Form einer anderen Komponente oder ein gesetztes *requestRun*-Flag (siehe Abbildung 4-13). Im Beispiel ist das Flag *requestRun* gesetzt (symbolisiert durch den ausgefüllten Kreis), sodass dieser Dienst in den Zustand RUNNING wechselt (6).

Ausgelöst wird eine Rekonfiguration innerhalb einer Komponente immer dann, wenn sie entweder Dienste mit gesetztem *requestRun*-Flag anbietet, oder Dienste anbietet, die von anderen Komponenten verwendet werden möchten. In diesem Fall legt der Komponentenentwickler für seine Komponenten fest, für welche Dienste das Flag gesetzt wird, und für welche nicht. Jedoch kann es so vorkommen, dass eine Komponente im Rahmen einer Anwendung verwendet werden soll, in der sehr wohl die Ausführung bestimmter Dienste sinnvoll ist, obwohl das Flag nicht gesetzt ist. Bietet eine Komponente beispielsweise einen Pulsdienst, über den der aktuelle Puls abgefragt werden kann, würde man das *requestRun*-Flag meist nicht setzen, da er keine grafische Nutzungsschnittstelle realisiert. Erzeugt der Dienst allerdings zusätzlich eine Logdatei mit dem Pulsverlauf, und möchte man in einer Anwendung diese Logdatei erzeugen und später beispielsweise manuell auswerten, so ist eine Ausführung des Dienstes für diese Anwendung sinnvoll, obwohl das Flag nicht gesetzt ist. An dieser Stelle fehlt die Möglichkeit, diesen Wert durch den Anwendungsentwickler überschreiben zu können. Durch die Möglichkeit der Spezifikation von Anwendungen (siehe Kapitel 6) wurde dieser Aspekt allerdings berücksichtigt.

Allerdings existieren auch Situationen, in denen Dienste ausgeführt werden, obwohl sie nicht verwendet werden. In Abbildung 8-3 ist eine derartige Situation dargestellt.

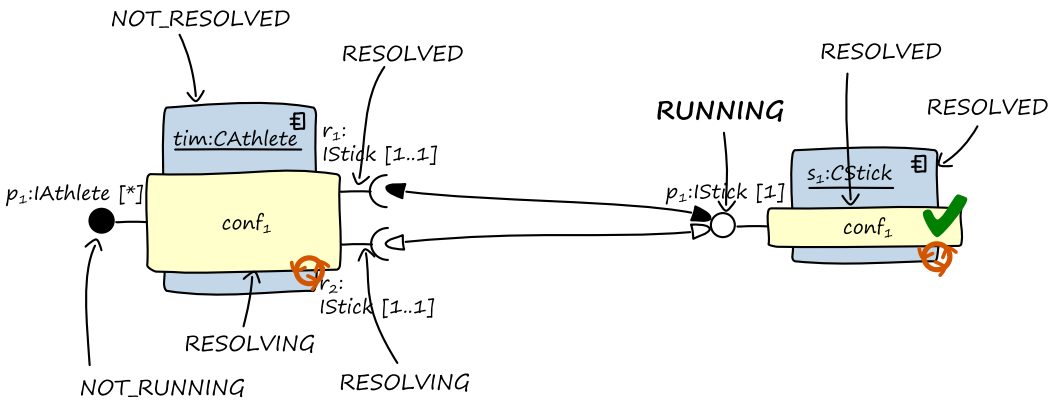


Abbildung 8-3: Dienst wird ausgeführt, obwohl er noch nicht verwendet wird

In diesem Fall benötigt die Athletenkomponente zwei Skistockkomponenten, von denen eine bereits verfügbar ist. Der entsprechende Dienst wurde im Laufe des vorangegangenen Konfigurationsprozesses von der Athletenkomponente zunächst angefragt, woraufhin der Dienst gestartet und anschließend der Komponente zugeteilt wurde. Allerdings kann die Athletenkomponente seinen Dienst erst dann ausführen, wenn ein zweiter Skistockdienst zur Verfügung steht. Und solange das nicht der Fall ist, wird der erste Skistockdienst ausgeführt, ohne dass er durch die Athletenkomponente tatsächlich verwendet wird.

8.2 Szenario 2 – Auflösen von Abhängigkeiten

Ziel dieses Szenarios ist die Evaluation der vorgestellten Lösung hinsichtlich der Möglichkeiten, für eine selbstorganisierende Komponente Abhängigkeiten zu Diensten anderer Komponenten zu spezifizieren und durch das Framework automatisiert auflösen zu lassen.

8.2.1 Anforderungen

Viele Komponenten benötigen Dienste anderer Komponenten um ihrerseits Dienste anbieten zu können. Selbstorganisierende Komponenten sind hierbei selbst dafür verantwortlich, diese Abhängigkeiten aufzulösen. Hierzu gehört insbesondere die Aufgabe, stetig die Verfügbarkeit von Diensten im System zu überwachen und ggf. auf das Hinzukommen und Wegfallen von Diensten zu reagieren. Insbesondere in mobilen, verteilten Anwendungen ist es nicht praktikabel, jeweils manuell auf derartige Ereignisse zu reagieren. So soll sich eine Navigationskomponente automatisch mit einer GPS-Komponente verbinden, wenn eine verfügbar ist. Ebenso sollte die Anwendung automatisch reagieren, wenn diese Komponente wieder aus dem System entfernt wird. Da die Realisierung einer solchen Funktionalität aufwändig und fehleranfällig ist, ist es sinnvoll, diese Aufgaben einem Framework zu überlassen. Auf diese Weise kann sich ein Komponentenentwickler auf die Realisierung der fachlichen Funktionalität konzentrieren, während das Framework für die Auflösung von Abhängigkeiten zu Diensten anderer Komponenten kümmert.

Damit das Framework diese Aufgabe übernehmen kann, muss es wissen, welche Abhängigkeiten für die Komponente existieren und welche Anforderungen die Komponente an Dienste anderer Komponenten stellt. Hierbei wurden zwei wesentliche Anforderungen identifiziert. Zunächst müssen Komponentenentwickler die Möglichkeit besitzen, dem Framework mitzuteilen, für welche Domänenschnittstellen sie Implementierungen durch Dienste anderer Komponenten benötigen. So muss für eine Navigationskomponente definiert werden können, dass ein Dienst

benötigt wird, welcher die Domänenschnittstelle *IGPS* realisiert. Des Weiteren ist in vielen Fällen auch die Anzahl der benötigten Dienste relevant. So wird für eine Navigationskomponente beispielsweise meist nur maximal ein GPS-Dienst benötigt. Obwohl also ggf. zahlreiche Dienste zur Verfügung stehen, sollte das Framework in diesem Fall die Navigationskomponente nur mit einem GPS-Dienst verbinden.

8.2.2 Umsetzung und Bewertung

In der hier vorgestellten Lösung werden Abhängigkeiten zu Diensten anderer Komponenten mit Hilfe sogenannter *RequiredServiceReferenceSets* definiert (siehe Abschnitt 4.7). Jedes *RequiredServiceReferenceSet* definiert hierbei eine Abhängigkeit zu einer Menge von Diensten, die alle dieselbe Domänenschnittstelle implementieren. Diese Domänenschnittstelle kann vom Komponentenentwickler je *RequiredServiceReferenceSet* mit Hilfe des Attributs *refersTo* festgelegt werden. Des Weiteren besteht die Möglichkeit, die minimale und maximale Anzahl von Referenzen auf benötigte Dienste zu spezifizieren. Hierzu definiert jedes *RequiredServiceReferenceSet* die Attribute *minNoOfRequiredRefs* und *maxNoOfRequiredRefs*, welche durch den Komponentenentwickler mit den gewünschten Werten belegt werden können.

Anhand eines Beispiels wird im Folgenden die Funktionsweise des Frameworks erläutert. Hierzu sei angenommen, dass eine Athletenkomponente existiert, die genau einen Dienst benötigt, der den aktuellen Puls des Sportlers zurückliefert. Die Komponente ist in Abbildung 8-4 dargestellt.

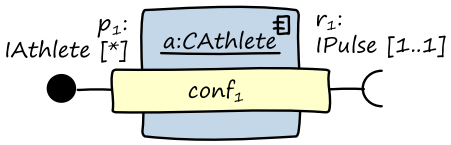


Abbildung 8-4: Komponente mit einem *RequiredServiceReferenceSet*

In diesem Fall sind die relevanten Attribute für das *RequiredServiceReferenceSet* also wie folgt belegt:

```
refersTo(r1) = IPulse
minNoOfRequiredRefs = 1
maxNoOfRequiredRefs = 1
```

Das Framework muss nun dafür sorgen, dass für dieses *RequiredServiceReferenceSet* genau eine Referenz auf einen Dienst bereitgestellt wird, der die Domänenschnittstelle *IPulse* implementiert. In Abbildung 8-5 sind die Konfigurationsschritte dargestellt, die ausgeführt werden, sobald ein entsprechender Dienst im System verfügbar ist.

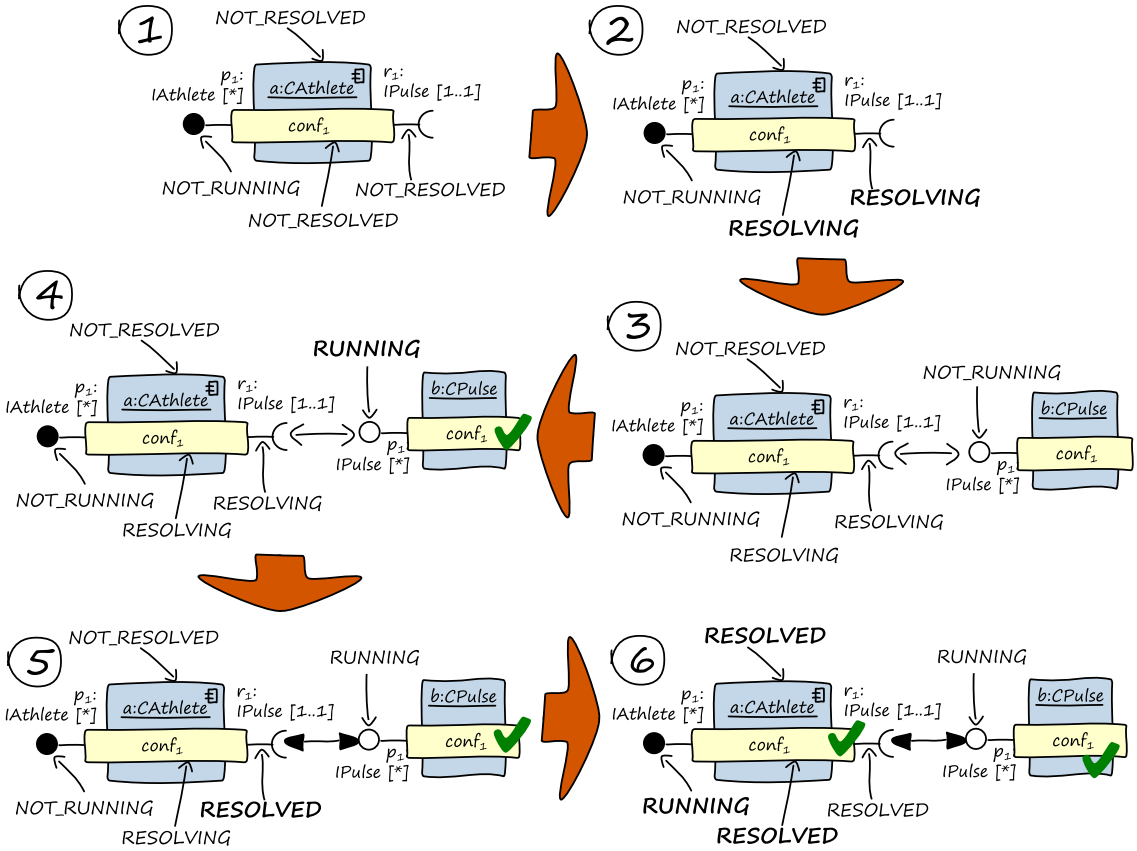


Abbildung 8-5: Beispiel zur Auflösung von Abhängigkeiten

Nachdem die Athletenkomponente im System installiert wurde, läuft ein Prozess ab, der bereits im vorangegangenen Abschnitt evaluiert wurde. Das *RequiredServiceReferenceSet* der Athletenkomponente befindet sich anschließend im Zustand RESOLVING (2).

Immer dann, wenn ein neuer Dienst im System installiert wird, wird er allen im System vorhandenen *RequiredServiceReferenceSets* bekannt gemacht. Stimmt die implementierte Domänenschnittstelle mit derjenigen überein, die vom *RequiredServiceReferenceSet* referenziert wird, so wird der Dienst in die Menge *canUse* des *RequiredServiceReferenceSets* aufgenommen. Befindet sich ein *RequiredServiceReferenceSet* im Zustand RESOLVING, so wird zusätzlich eine Nutzungsanfrage durch Aufruf des Triggers *wantsUse* an den neuen Dienst gestellt und der Dienst in die Menge *wantsUse* aufgenommen (3). Dieser Aufruf hat zur Folge, dass für die Pulskomponente wiederum der bereits bekannte Konfigurationsprozess zum Starten des Pulsdienstes durchgeführt wird. Das Ergebnis ist in Schritt 4 der Abbildung 8-5 dargestellt.

Beim Übergang in den Zustand RUNNING wird denjenigen *RequiredServiceReferenceSets* der Dienst zur Nutzung bereitgestellt, die eine Nutzungsanfrage gestellt haben. Die Zuordnung des Dienstes geschieht durch Aufruf des Triggers *serviceAssigned*. Damit sind für das *RequiredServiceReferenceSet* alle Anforderungen erfüllt und es kann in den Zustand RESOLVED wechseln (5). Abschließend wechseln dann sowohl die *ComponentConfiguration* als auch die *DynamicAdaptiveComponent* in den Zustand RESOLVED und der Athletendienst in den Zustand RUNNING (6).

Ein häufig auftretender Fall ist der, dass mehr Dienste bereitstehen, als tatsächlich von einem *RequiredServiceReferenceSet* benötigt werden. In diesem Fall muss das Framework sicherstellen,

dass maximal die Anzahl an Diensten verwendet wird, die im Attribut *maxNoOfRequiredRefs* hinterlegt ist. Zur Evaluation des Verhaltens sei angenommen, dass ausgehend vom Ergebnis des Konfigurationsprozesses in Abbildung 8-5 eine weitere Pulskomponente dem System hinzugefügt wird. Der resultierende Konfigurationsprozess ist in Abbildung 8-6 dargestellt.

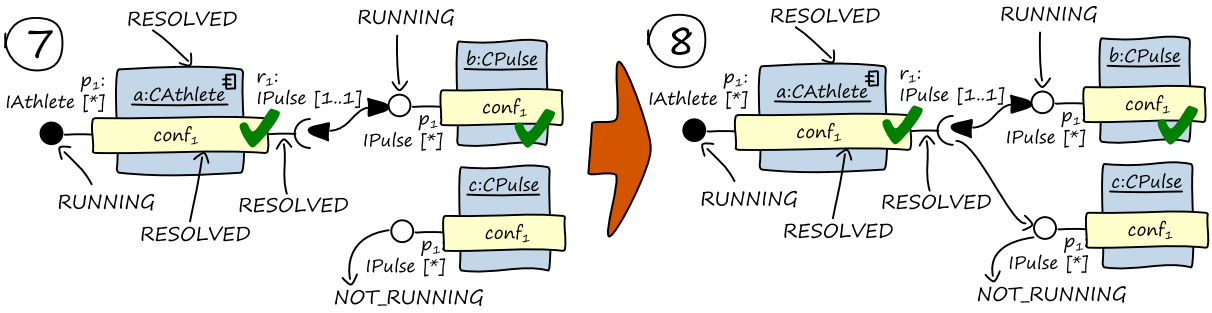


Abbildung 8-6: Es stehen mehr Dienste zur Verfügung als benötigt

Befindet sich ein *RequiredServiceReferenceSet* im Zustand RESOLVED und kommt ein neuer Dienst hinzu, so wird dieser Dienst zunächst in die Menge *canUse* aufgenommen (8). Eine Nutzungsanfrage wird allerdings nur dann gestellt, wenn die maximale Anzahl benötigter Dienste noch nicht erreicht ist (siehe Abbildung 4-16). Dies ist im Beispiel nicht der Fall, so dass keine weiteren Konfigurationsschritte durchgeführt werden.

Zusammenfassend lässt sich sagen, dass das Konzept der *RequiredServiceReferenceSets* geeignet ist, Basisanforderungen hinsichtlich benötigter Dienste zu formulieren. Hierbei können Anforderungen hinsichtlich der implementierten Domänenschnittstelle sowie der minimalen und maximalen Anzahl benötigter Dienste spezifiziert werden. Zudem ist das Framework in der Lage, diese Anforderungen zuverlässig, automatisiert und zur Laufzeit umzusetzen.

8.3 Szenario 3 – Berücksichtigung erweiterter Anforderungen bei der Auflösung von Abhängigkeiten

Ziel dieses Szenarios ist die Evaluation der Lösung hinsichtlich der Möglichkeiten, Anforderungen an benötigte Dienste spezifizieren zu können, welche über die Angabe einer Domänenschnittstelle sowie der minimalen und maximalen Anzahl benötigter Dienste hinausgeht. Außerdem wird die Lösung dahingehend evaluiert, inwiefern diese Spezifikationen automatisiert durch das Framework berücksichtigt werden.

8.3.1 Anforderungen

In einigen Fällen reicht es nicht aus, lediglich Zugriff auf eine beliebige Realisierung einer Domänenschnittstelle durch eine andere Komponente zu haben. So kann es für eine Komponente entscheidend sein, dass verwendete Dienste beispielsweise bestimmte Qualitätskriterien hinsichtlich Antwortzeit oder Genauigkeit von Berechnungsresultaten einhalten. Für eine Navigationskomponente wäre z.B. ein GPS-Dienst nur dann von Nutzen, wenn dieser Dienst die Position mit einer genügend großen Genauigkeit zurückliefert. Ein Framework muss hierbei dem Komponentenentwickler die Möglichkeit bieten, derartige Anforderungen hinterlegen zu können. Das Framework muss dann zur Laufzeit die Abhängigkeiten unter Berücksichtigung der Angaben auflösen. Zusätzlich muss das Framework in der Lage sein, regelmäßig zu prüfen, ob die definierten Anforderungen noch eingehalten

werden. Denn häufig unterliegt z.B. die Qualität angebotener Dienste Schwankungen. So kann ein GPS-Sensor zu einem Zeitpunkt sehr genaue Positionsangaben liefern, etwas später jedoch nur noch sehr ungenaue Angaben (beispielsweise bei einer Tunneldurchfahrt). Hierauf muss das Framework angemessen reagieren, z.B. durch Austausch des verwendeten Dienstes.

8.3.2 Umsetzung und Bewertung

Eine Schnittstellenrolle definiert eine Menge von Diensten, die bestimmte Anforderungen erfüllen. Diese Anforderungen werden innerhalb der Schnittstellenrolle mit Hilfe des Prädikats *isConformTo* definiert. Dieses Prädikat bekommt als Parameter einen Dienst übergeben und liefert dann *true* zurück, wenn dieser alle Anforderungen erfüllt. Hierbei werden Schnittstellenrollen im Domänenmodell hinterlegt. Die vorgestellte Lösung sieht vor, dass Komponentenentwickler für *RequiredServiceReferenceSets* jeweils eine Schnittstellenrolle referenzieren können. Das Framework sorgt zur Laufzeit dafür, dass nur diejenigen Dienste einem *RequiredServiceReferenceSet* zugewiesen werden, welche die Anforderungen der referenzierten Schnittstellenrolle erfüllen. Außerdem überprüft das Framework zyklisch, ob die verwendeten Dienste die Anforderungen der referenzierten Schnittstellenrolle erfüllen. Ist dies nicht der Fall, so wird versucht, einen alternativen Dienst zu finden. Gelingt dies nicht, und können die Anforderungen des *RequiredServiceReferenceSets* nicht mehr erfüllt werden, so werden die verbundenen *ComponentConfigurations* hierüber benachrichtigt und ggf. die Ausführung angebotener Dienste gestoppt.

Im Folgenden wird anhand eines kleinen Beispiels die Funktionsweise des Frameworks evaluiert. Hierzu werden zum einen die Schnittstellenrollen, welche bereits in Abschnitt 5.5 vorgestellt und in Formel 8-1 nochmals aufgeführt sind, herangezogen:

$$\textbf{context}\ LeftStickRole :: isConformTo(IStick\ ps): bool$$
$$\textbf{post}: result = ps.getSide() = 'left'$$

$$\textbf{context}\ RightStickRole :: isConformTo(IStick\ ps): bool$$
$$\textbf{post}: result = ps.getSide() = 'right'$$

Formel 8-1

Außerdem wird eine Komponente betrachtet, welche zwei *RequiredServiceReferenceSets* definiert, die jeweils eine Schnittstellenrolle referenzieren. Diese Komponente ist in Abbildung 8-7 dargestellt.

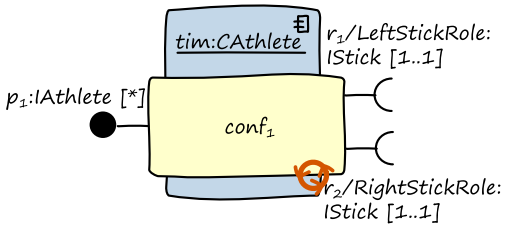


Abbildung 8-7: Komponente, die zwei Schnittstellenrollen referenziert

Der angebotene Dienst p_1 benötigt zwei Dienste, um ausgeführt zu werden. Beide müssen die Domänenschnittstelle *IStick* implementieren, wobei für r_1 ein Dienst benötigt wird, der die Schnittstellenrolle *LeftStickRole* realisiert, und für r_2 einer, der die Schnittstellenrolle

RightStickRole umsetzt. Im Folgenden wird nun untersucht, wie das Framework diese Anforderungen zur Laufzeit automatisiert berücksichtigt.

Nach Installation der Komponente und Durchführung der bereits vorgestellten Konfigurationsschritte befinden sich die einzelnen Komponentenbestandteile in den Zuständen, wie sie in Abbildung 8-8 dargestellt sind.

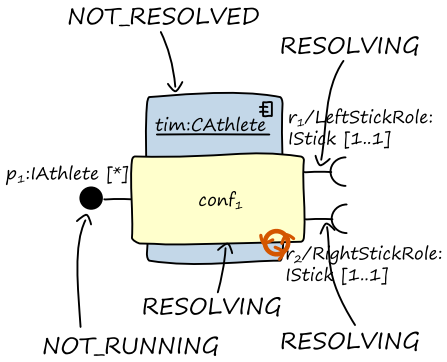


Abbildung 8-8: Zustände der Athletenkomponente nach Ausführung der ersten Konfigurationsschritte

Die Komponente befindet sich nach Installation in einem Zustand, in dem sie auf geeignete Dienste für die beiden *RequiredServiceReferenceSets* wartet. Es sei nun angenommen, dass eine Komponente installiert wird, die die Schnittstelle *IStick* implementiert. Daraufhin nehmen beide *RequiredServiceReferenceSets* den neuen Dienst in die Menge *canUse* auf. Gleichzeitig melden sie durch Aufruf von *wantsUse* ein Nutzungsinteresse an und der Dienst wird zusätzlich der Menge *wantsUse* hinzugefügt. Das vorläufige Resultat ist in Abbildung 8-9 dargestellt.

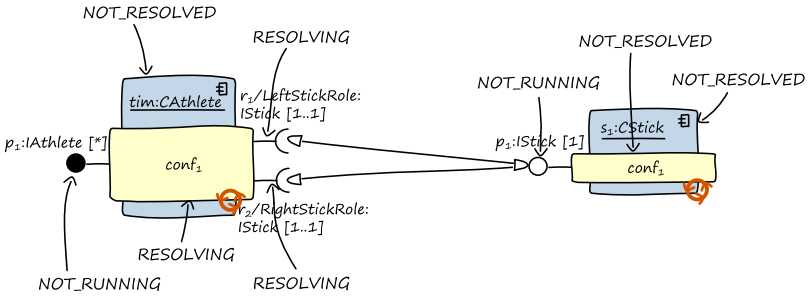


Abbildung 8-9: Situation nach Installation einer Skistockkomponente

Der Aufruf des Triggers beim Skistockdienst führt innerhalb der Skistockkomponente zu einer Reihe von Konfigurationsschritten, die bereits zuvor evaluiert wurden. Das Resultat ist, dass der Skistockdienst schließlich in den Zustand RUNNING wechselt, und somit zur Verwendung bereitsteht. Des Weiteren sei angenommen, dass der Dienst bei Aufruf der Methode *getSide* den Wert „left“ zurückgibt. Das Zwischenergebnis ist in Abbildung 8-10 dargestellt.

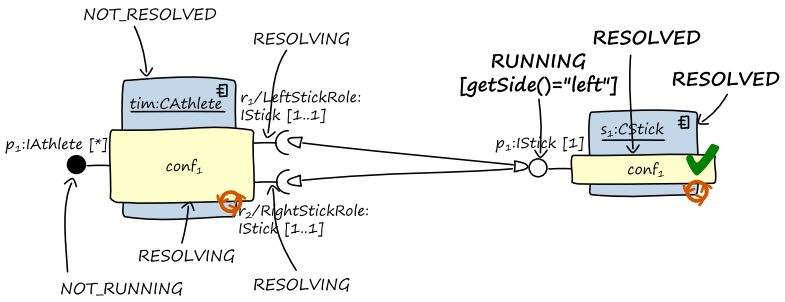


Abbildung 8-10: Die Skistockkomponente kann verwendet werden

An dieser Stelle erfolgt nun die Zuteilung des Dienstes an diejenigen *RequiredServiceReferenceSets*, welche ein Nutzungsinteresse angemeldet haben. Allerdings kann der Dienst in diesem Fall nur von maximal einer anderen Komponente verwendet werden, da das Attribut *maxNoOfUsers* des Dienstes auf den Wert 1 gesetzt wurde. Das Framework geht nun wie folgt vor. Beim Übergang in den Zustand RUNNING wird für jedes *RequiredServiceReferenceSet* der Menge *runRequestedBy* geprüft, ob der Dienst die Anforderungen derjenigen Schnittstellenrolle umsetzt, die vom *RequiredServiceReferenceSet* referenziert wird (siehe Abschnitt 5.4.1). Ist die Obergrenze an Nutzern erreicht, wird keine weitere Zuteilung mehr vorgenommen. Im Beispiel würde somit eine Zuordnung des Dienstes zum *RequiredServiceReferenceSet* r_1 stattfinden. Das Resultat des Konfigurationsprozesses, der durch Hinzufügen der Skistockkomponente ausgelöst wurde, ist in Abbildung 8-11 dargestellt.

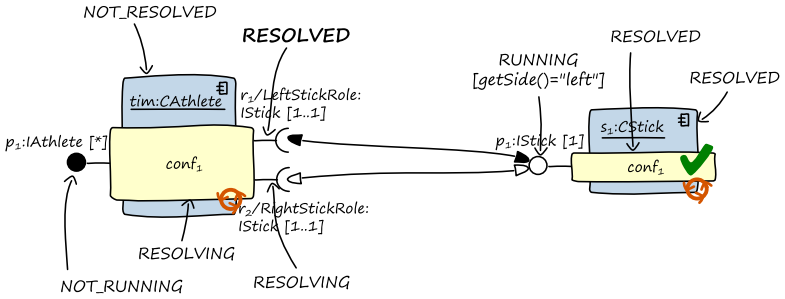


Abbildung 8-11: Zuordnung einer rollenkompatiblen Skistockkomponente

Das Framework der Skistockkomponente prüft nun regelmäßig, ob der Dienst kompatibel zu derjenigen Schnittstellenrolle ist, die von r_1 referenziert wird (siehe Abschnitt 5.4.4). Sobald erkannt wird, dass dies nicht mehr der Fall ist, wird dem *RequiredServiceReferenceSet* der Dienst wieder entzogen.

Kommt nun ein Dienst hinzu, der einen rechten Skistock repräsentiert, so ergibt sich hieraus nach Abschluss aller resultierenden Konfigurationsschritte die in Abbildung 8-12 dargestellte Situation.

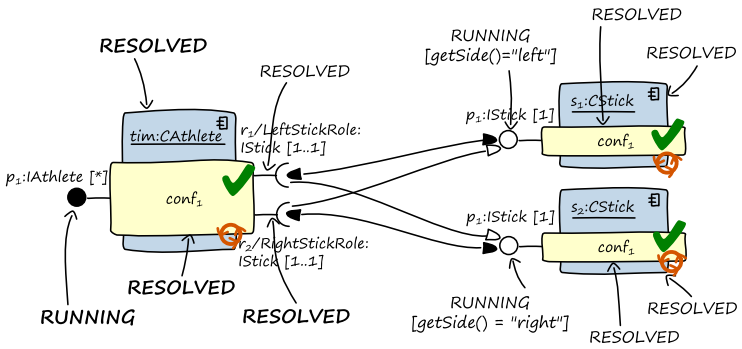


Abbildung 8-12: Situation nach Hinzufügen eines rechten Skistocks

Der Athletendienst kann in dieser Situation ausgeführt werden, da nun die Anforderungen beider *RequiredServiceReferenceSets* erfüllt sind. Hierbei besteht die Menge *canUse* des *RequiredServiceReferenceSets* r_1 aus zwei Elementen, nämlich zum einen aus dem verwendeten Dienst p_1 und zum anderen aus p_2. Sobald für r_1 der Dienst p_1 nicht mehr zur Verfügung steht, wird das *RequiredServiceReferenceSet* eine Nutzungsanfrage an alle Dienste der Menge *canUse* stellen. In dieser Menge werden all diejenigen Dienste verwaltet, die die passende Schnittstelle implementieren. In der dargestellten Situation liegen keinerlei Nutzungsanfragen vor, da beide *RequiredServiceReferenceSets* bereits die maximale Anzahl benötigter Dienste zur Verfügung haben. Sobald dies der Fall ist, werden sämtliche noch ausstehende Nutzungsanfragen vom jeweiligen *RequiredServiceReferenceSet* zurückgezogen.

8.4 Szenario 4 – Aktivierung der bestmöglichen Komponentenkonfiguration

Ziel des Szenarios ist die Evaluation des Konzeptes der *ComponentConfigurations*, welches in Kapitel 4 eingeführt wurde. Zur Evaluation werden nun zunächst diejenigen Anforderungen an mobile, verteilte Systeme erläutert, die die Einführung dieses Konzeptes begründen. Im Anschluss wird dessen Umsetzung erläutert und evaluiert.

8.4.1 Anforderungen

Viele Komponenten bieten nicht nur einen Dienst an, sondern häufig mehrere Dienste. Allerdings besitzen diese Dienste meist unterschiedliche Anforderungen hinsichtlich der Nutzung von Diensten anderer Komponenten. So bieten Komponenten häufig bestimmte Basisdienste an, die auch dann ausgeführt werden können, wenn keinerlei Funktionalität anderer Komponenten zur Verfügung steht. Andere, ggf. leistungsfähigere Dienste der Komponente, benötigen wiederum Zugriff auf bestimmte Dienste anderer Komponenten. So könnte eine Navigationskomponente einen Basisdienst anbieten, der lediglich eine Karte anzeigt. Sobald dieser Komponente dann ein GPS-Dienst zur Verfügung stellt, könnte die Komponente dann einen weiteren Dienst anbieten, der eine Route berechnen kann.

Insbesondere in mobilen verteilten Systemen wäre es nicht praktikabel, derartige Rekonfigurationen manuell durch den Endanwender vornehmen zu lassen. Vielmehr müssen derartige Reaktionen und Anpassungen automatisiert erfolgen. Hierbei kann kein allgemeingültiges Kriterium angegeben werden, welche mögliche Konfiguration jeweils die Beste ist. Hier ist es also Aufgabe des Komponentenentwicklers, eine entsprechende Ordnung festzulegen.

8.4.2 Umsetzung und Bewertung

Die im Rahmen der Arbeit präsentierte Lösung führt zur Realisierung der Anforderungen das Konzept der *ComponentConfigurations* ein. Jede *ComponentConfiguration* bildet eine Menge von angebotenen Diensten einer Komponente auf eine Menge von benötigten Diensten (*RequiredServiceReferenceSets*) ab. Das entwickelte Framework ermöglicht hierbei zum einen die Definition beliebig vieler *ComponentConfigurations,* und zum anderen die Definition einer Ordnung über diesen Konfigurationen. Das Framework sorgt zur Laufzeit automatisiert dafür, dass stets die bestmögliche Konfiguration aus Sicht der Komponente realisiert wird.

Anhand des in Abbildung 8-13 dargestellten Beispiels werden die Abläufe zur Realisierung dieser Funktionalität evaluiert.

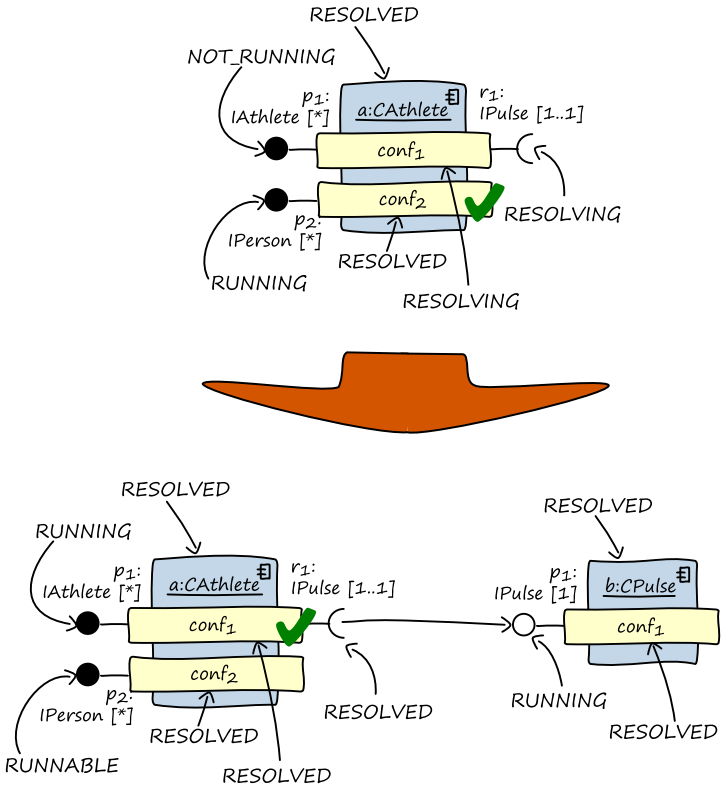


Abbildung 8-13: Beispiel zur Berücksichtigung von *ComponentConfiguration*-Prioritäten

In der Ausgangssituation wird hierbei angenommen, dass im System eine Komponente mit zwei *ComponentConfigurations* installiert ist. Diese wurde bereits durch das Framework derart konfiguriert, dass keine weiteren Konfigurationsschritte mehr durchgeführt werden müssen. In diesem Fall konnte zunächst nur die zweitbeste Konfiguration aktiviert werden, da die Abhängigkeiten der bestmöglichen Konfiguration noch nicht aufgelöst werden konnten. Hierzu fehlt eine Komponente, welche einen Dienst anbietet, der die Domänenschnittstelle *IPulse* implementiert. Diese wird im Szenario hinzugefügt und anhand dessen evaluiert, wie das Framework automatisiert die bestmögliche Konfiguration und die betreffenden Dienste startet bzw. stoppt.

Während der Installation einer neuen Komponente werden im ersten Schritt alle *RequiredServiceReferenceSets* aller bislang im System installierten Komponenten über das Hinzukommen neuer Dienste informiert durch Aufruf von *newService* informiert. So wird im

Beispiel das *RequiredServiceReferenceSet* der Athletenkomponente über das Hinzukommen des Pulsdienstes informiert. Sämtliche Zustandsautomaten der neuen Komponente verbleiben hierbei zunächst in ihrem Initialzustand, da weder ein Nutzungswunsch für den angebotenen Pulsdienst vorliegt, noch das Flag *requestRun* gesetzt ist. Die Situation stellt sich somit wie in Abbildung 8-14 angegeben dar.

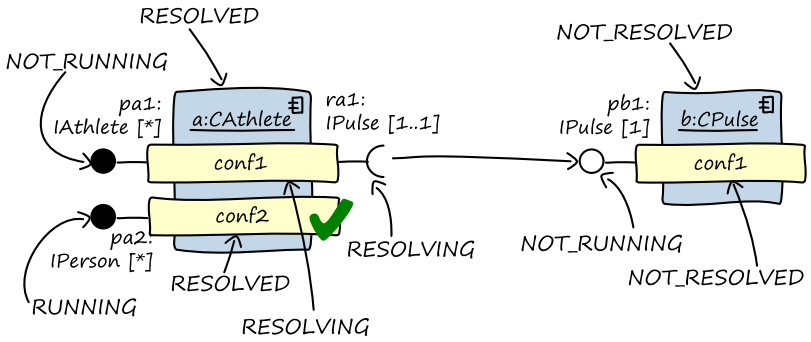


Abbildung 8-14: Situation unmittelbar nach Installation der Pulskomponente

Das *RequiredServiceReferenceSet* der Athletenkomponente reagiert auf das Hinzukommen des neuen Pulsdienstes, indem an diesen eine Nutzungsanfrage durch Aufruf von *wantsUse* gestellt wird (siehe Abbildung 4-16). Das *RequiredServiceReferenceSet* wartet daraufhin auf die Zuteilung des Dienstes, also auf den Aufruf *serviceAssigned* durch den Pulsdienst. Die Nutzungsanfrage beim Pulsdienst setzt innerhalb der Pulskomponente den bereits im Abschnitt zuvor diskutierten Konfigurationsprozess in Gang. Im Gegensatz zum vorherigen Szenario ist der Trigger hier allerdings nicht das gesetzte *requestRun*-Flag innerhalb des Pulsdienstes, sondern eine Nutzungsanfrage durch eine andere Komponente. Das Ergebnis des internen Konfigurationsprozesses der Pulskomponente ist in Abbildung 8-15 dargestellt.

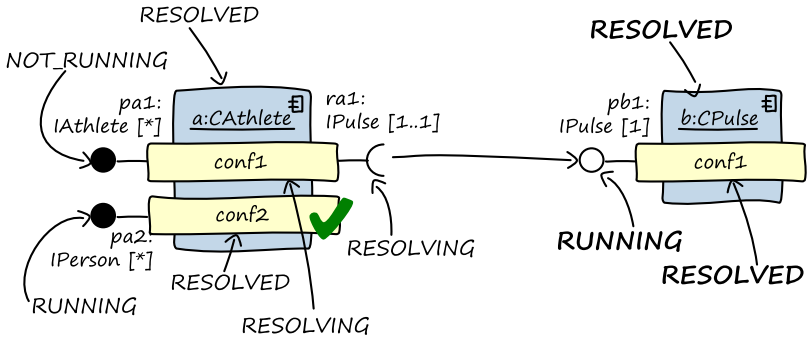


Abbildung 8-15: Situation nach Abschluss des komponenteninternen Konfigurationsprozesses der Pulskomponente

Beim Übergang des Pulsdienstes in den Zustand RUNNING erfolgt gleichzeitig die Zuteilung des an diejenigen *RequiredServiceReferenceSets*, die eine Nutzungsanfrage gestellt haben. Dies geschieht durch Aufruf der Methode *serviceAssigned* beim *RequiredServiceReferenceSet*. Dieses kann daraufhin in den Zustand RESOLVED wechseln und die *ComponentConfiguration* darüber informieren, dass die Abhängigkeit aufgelöst werden konnte. Daraus ergibt sich die in Abbildung 8-16 dargestellte Situation.

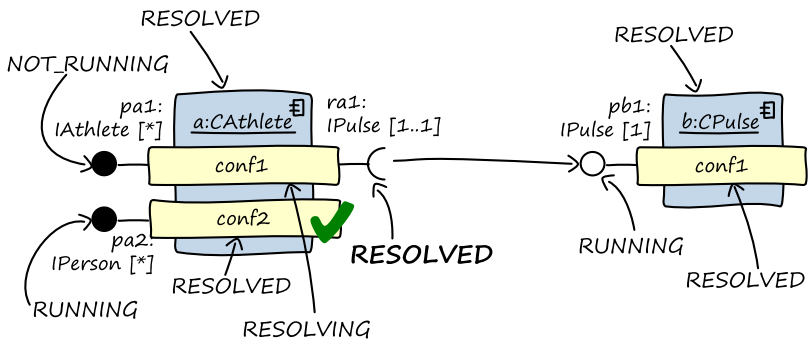


Abbildung 8-16: Die Situation nach Zuweisung des Pulsdienstes an die Athletenkomponente

Da die *ComponentConfiguration config1* der Athletenkomponente lediglich eine Abhängigkeit zu Diensten anderer Komponenten definiert und diese bereits aufgelöst werden konnte, wechselt sie in den Zustand RESOLVED und teilt dies der *DynamicAdaptiveComponent* durch Aufruf des Triggers *configurationResolved* mit. Die resultierende Situation ist in Abbildung 8-17 dargestellt.

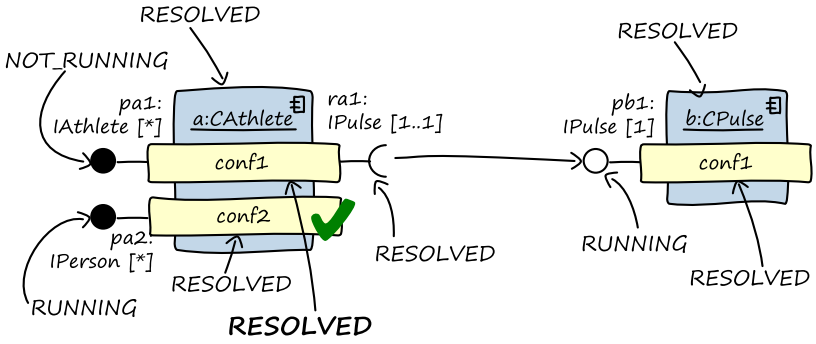


Abbildung 8-17: Die Athletenkomponente besitzt nun zwei aktivierbare *ComponentConfigurations*

Der Athletenkomponente stehen nun zwei aktivierbare *ComponentConfigurations* zur Verfügung, wobei in der dargestellten Situation noch die schlechtere von beiden aktiv ist. Aufgabe des Frameworks ist es nun, die bessere von beiden zu aktivieren. Hierzu muss zum einen der Dienst *pa2* der *ComponentConfiguration config2* gestoppt, und der Dienst *pa1* der *ComponentConfiguration config1* gestartet werden. Realisiert wird diese Funktionalität innerhalb des Frameworks beim Aufruf des Triggers *configurationResolved* bei der *DynamicAdaptiveComponent*. Hierbei wird dem Dienst *pa1* durch Aufruf von *serviceRunnable* mitgeteilt, dass der Dienst ausgeführt werden kann, und dem Dienst *pa2* durch Aufruf von *serviceNotRunnable* mitgeteilt, dass eine Ausführung nicht mehr möglich ist. Das Resultat des Konfigurationsprozesses ist in Abbildung 8-18 nochmals dargestellt.

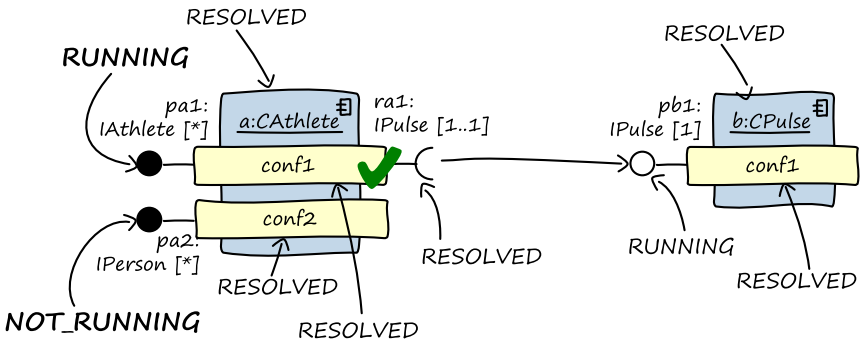


Abbildung 8-18: Ergebnis des Konfigurationsprozesses nach Hinzufügen der Pulskomponente

Die Aktivierung der bestmöglichen *ComponentConfiguration* einer Komponente basiert darauf, dass immer dann, wenn eine *ComponentConfiguration* aktivierbar wird, oder nicht mehr aktivierbar ist, eine Überprüfung stattfindet. Sobald eine im Vergleich zur aktuell aktiven *ComponentConfiguration* bessere aktivierbar geworden ist, werden die betroffenen angebotenen Dienste gestartet bzw. gestoppt. Dies kann wiederum Auswirkungen auf Komponenten haben, die diese Dienste benötigen.

8.5 Szenario 5 – Berücksichtigung Anwendungsarchitektur-spezifischer Anforderungen

Ziel dieses Szenarios ist die Evaluation des Frameworks hinsichtlich der Möglichkeiten, derart Einfluss auf den Konfigurationsprozess zu nehmen, so dass komponentenübergreifende, anforderungsspezifische Anforderungen berücksichtigt werden.

8.5.1 Anforderungen

Werden selbstorganisierende Systeme basierend auf selbstorganisierenden Komponenten erstellt, so werden in erster Linie Anforderungen berücksichtigt, welche durch einzelne Komponenten definiert werden. So werden Komponenten versuchen, Abhängigkeiten zu Diensten anderer Komponenten derart aufzulösen, dass komponentenspezifische Anforderungen berücksichtigt werden. Allerdings führt dieses Vorgehen nicht in jedem Fall auch zu einem optimalen bzw. gewünschten Gesamtsystem. Eine Anforderung an ein Framework zur Entwicklung und Ausführung verteilter, mobiler Systeme ist es deshalb, anwendungsspezifische Anforderungen in den Konfigurationsprozess einfließen lassen zu können. Hierbei sollen allerdings die Vorteile des Einsatzes selbstorganisierender Komponenten möglichst erhalten bleiben. Hierzu zählen insbesondere die Beibehaltung einer leichtgewichtigen Infrastruktur und der Verzicht auf eine komplexe zentrale Konfigurationseinheit.

Gerade in mobilen und verteilten Anwendungen gibt es häufig die Situation, dass zur Laufzeit neue Komponenten und Dienste durch Drittanbieter bereitgestellt werden. Hierbei soll eine Integration dieser Komponenten in eine laufende Anwendung möglich sein, ohne die Anwendungsspezifikation anpassen zu müssen.

8.5.2 Umsetzung und Bewertung

Die in dieser Arbeit vorgestellte Lösung bietet einem Anwendungsentwickler sowohl die Möglichkeit, Anforderungen hinsichtlich der Komponentenauswahl als auch hinsichtlich der Verbindungen zwischen diesen Komponenten spezifizieren zu können. Des Weiteren sorgt das Framework zur Laufzeit für die Realisierung und Einhaltung der Anforderungen. Die Lösung wurde in Kapitel 6 ausführlich dargestellt. Die Funktionsweise der erarbeiteten Lösung wird im Folgenden anhand eines kleinen Szenarios evaluiert.

Hierzu sei angenommen, dass eine Anwendung erstellt werden soll, bei der ein Trainer die Pulsverläufe seiner Sportler angezeigt bekommt. Zudem soll es einen zweiten Trainer geben, der nur die Daten derjenigen Athleten angezeigt bekommt, die jünger sind als 16 Jahre. Und schließlich soll sichergestellt werden, dass sowohl Trainer als auch Athleten dem Sportverein „WSV“ angehören. Die in Abbildung 8-19 dargestellte Anwendungsspezifikation definiert diese Anforderungen mit Hilfe der im Rahmen der Arbeit vorgestellten Konzepte.

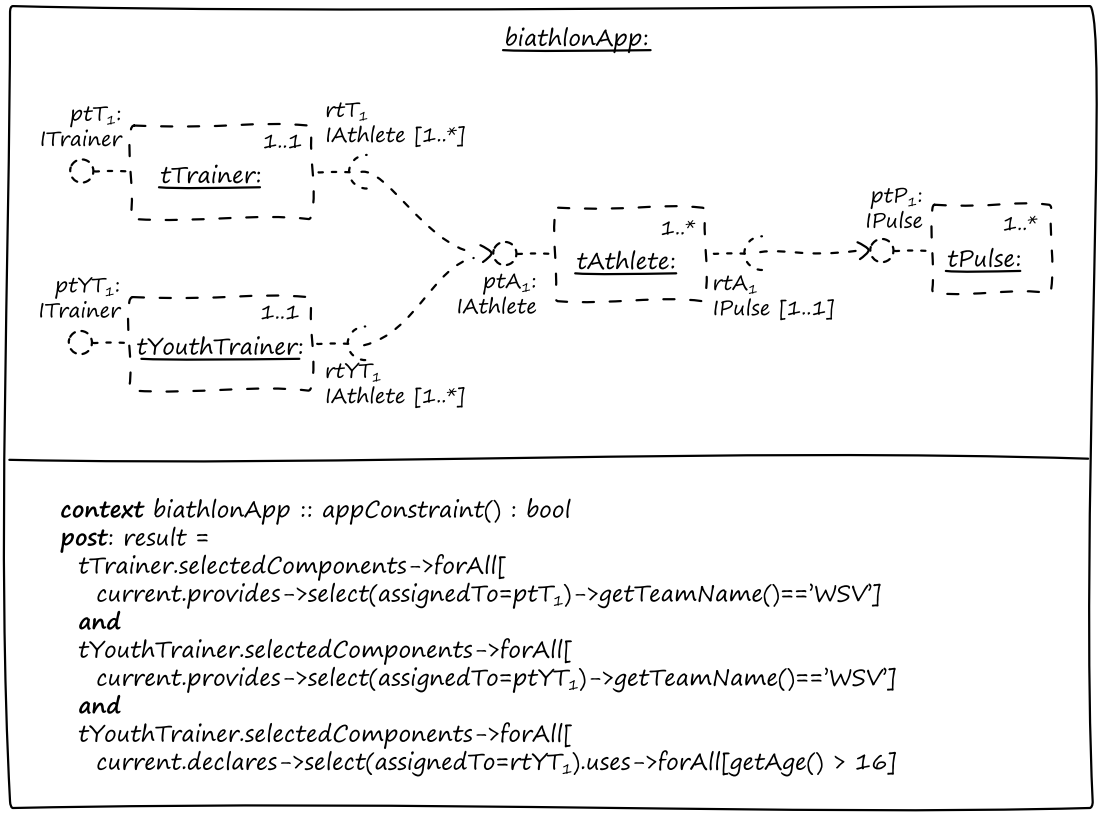


Abbildung 8-19: Anwendung zur Pulsüberwachung durch zwei Trainer

Die vorgestellte Lösung bietet einem Anwendungsentwickler zahlreiche Möglichkeiten, bestimmte Anforderungen an das zu konfigurierende System zu definieren und automatisiert umsetzen zu lassen. Die Mächtigkeit der Spezifikationsmöglichkeiten wird im Folgenden kurz umrissen.

Zum einen lässt die Lösung die Spezifikation von Anforderungen hinsichtlich der Struktur der Anwendung zu. Hierbei können sowohl Anforderungen an die Struktur der beteiligten Komponenten definiert werden, als auch an die Verbindungen zwischen diesen. Im Beispiel werden beispielsweise nur Trainerkomponenten aufgenommen, die eine *ComponentConfiguration* besitzen, welche die Schnittstelle *ITrainer* anbietet und ein *RequiredServiceReferenceSet* definieren, welches die Domänenschnittstelle *IAthlete* referenziert. Außerdem werden nur Pulskomponenten zugelassen, die die Domänenschnittstelle *IPulse* anbieten. Neben der Struktur einzelner Komponenten können auch Aussagen über die Anzahl der Komponenten gemacht werden, die für ein *Template* minimal und maximal benötigt werden. Erst wenn genügend Komponenten für jedes *Template* bereitstehen, kann die Anwendung ausgeführt werden. Mit Hilfe von Verbindungen zwischen den einzelnen *Templates* lassen sich zudem Anforderungen hinsichtlich der Verbindungen zwischen Komponenten spezifizieren. So ist im Beispiel aus Abbildung 8-19 vorgegeben, dass diejenigen Komponenten, welche den Trainer-*Templates* zugeordnet wurden, ausschließlich mit denjenigen Diensten verbunden werden dürfen, deren anbietende Komponente dem Athleten-*Template* zugeordnet wurden. Und auch hier lassen sich Vorgaben hinsichtlich der minimalen und maximalen Anzahl benötigter Referenzen durch den Anwendungsentwickler machen. Das Framework sorgt zur Laufzeit dafür, dass die einzelnen Komponenten diese Vorgaben einhalten.

Neben Anforderungen an die Struktur des Gesamtsystems lassen sich durch den Anwendungsentwickler auch Anforderungen hinsichtlich des Zustandes der beteiligten Komponenten machen, sowie komponentenübergreifende Anforderungen spezifizieren. Hierzu dient das Prädikat *appConstraint*. Dieses wird vom Anwendungsentwickler definiert und liefert dann *true* zurück, falls die erzeugte Konfiguration alle Anforderungen erfüllt (siehe Formel 6-17). Die Lösung sieht hierzu vor, zunächst eine Konfiguration zu erzeugen, welche alle strukturellen Anforderungen erfüllt, und bei der alle relevanten Dienste ausgeführt werden. Das Prädikat *appConstraint* wird erst anschließend ausgewertet. Auf diese Weise können auch diejenigen Anforderungen überprüft werden, welche sich auf den Zustand einzelner Komponenten beziehen. Die Anforderungen, die mit Hilfe des Prädikats formuliert werden können, sind vielfältig. So kann innerhalb des Prädikates auf die angebotenen Dienste derjenigen Komponenten zugegriffen werden, welche durch das Framework für eine Konfiguration ausgewählt wurden. Außerdem können Anforderungen hinsichtlich derjenigen Dienste spezifiziert werden, welche durch eine Komponente verwendet werden. Der Einstiegspunkt für die Spezifikation des Prädikats ist die Menge von *Templates*, welche durch die Assoziation *contains* adressiert werden kann. Über die Assoziation *selectedComponents* können innerhalb des Prädikats dann Eigenschaften definiert werden, die sämtliche Elemente dieser Menge erfüllen müssen, damit die Anwendung ausgeführt werden kann. Innerhalb der Komponente kann über die Assoziation *current* auf die aktive Konfiguration und den hiermit verbundenen *ProvidedServices* und *RequiredServiceReferenceSets* zugegriffen werden und Bedingungen hierüber formuliert werden. Im Beispiel aus Abbildung 8-19 wird beispielsweise gefordert, dass für die Trainerkomponenten der Anwendung gelten muss, dass bei Aufruf von *getTeamName()* der Wert „WSV“ zurückgeliefert wird. Erfüllt eine Konfiguration diese Bedingung nicht, liefert das Prädikat *false* zurück und es muss eine neue Konfiguration erzeugt werden.

Um eine gültige Konfiguration zu finden, erzeugt das Framework iterativ alle möglichen Konfigurationen, welche die strukturellen Anforderungen erfüllen. Sobald eine solche Konfiguration erzeugt wurde, wird das Prädikat *appConstraint* ausgewertet. Dieser Prozess wird solange fortgesetzt, bis eine Konfiguration gefunden wurde, bei der das Prädikat zu *true* ausgewertet wird.

Die Erzeugung neuer Konfigurationen sowie zur Überprüfung der Anforderungen wurde der Lösung eine zentrale Konfigurationseinheit hinzugefügt. Es wurde somit einer der Vorteile rein dezentral organisierter Systeme basierend auf ausschließlich selbstorganisierenden Komponenten aufgegeben. Allerdings erlaubt das hier vorgestellte Verfahren auch weiterhin ein weitestgehend selbstorganisierendes Verhalten der einzelnen Komponenten. Hierbei hat die zentrale Konfigurationseinheit lediglich die Aufgabe, die zur Verfügung stehenden Komponenten den geeigneten *Templates* zuzuordnen und den Komponenten diese Zuordnung mitzuteilen. Jede selbstorganisierende Komponente kann anschließend auf die strukturellen Bedingungen des *Templates* zugreifen und so z.B. die Menge derjenigen Dienste bestimmten, die zur Auflösung von Abhängigkeiten in Frage kommen. Für jede Komponente ergibt sich somit eine endliche Menge an möglichen Variationsmöglichkeiten. Die zentrale Konfigurationseinheit hat nach Zuordnung der Komponenten lediglich die Aufgabe, einzelnen Komponenten mitzuteilen, wenn sie eine nächste Variation erzeugen sollen. Hierzu realisiert jede Komponente im Grunde einen Iterator, der *false* bei Aufruf von *hasNext()* zurückliefert, sobald alle möglichen Variationen erzeugt wurden.

Für das Beispiel sei angenommen, dass die zentrale Konfigurationseinheit folgende initiale Zuordnung von Komponenten zu *Templates* vorgenommen hat (siehe Abbildung 8-20).

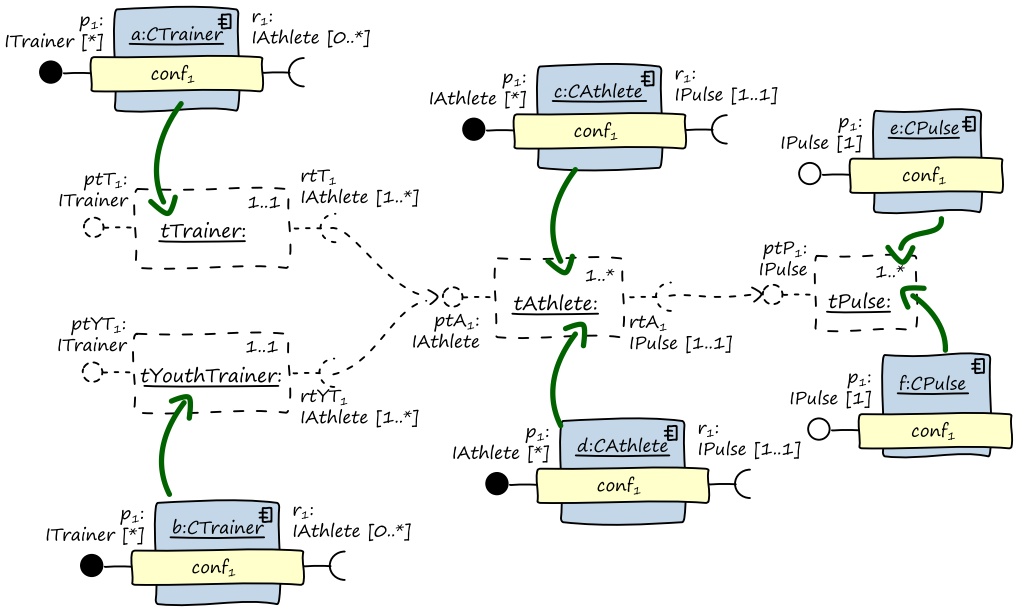


Abbildung 8-20: Komponenten des Beispiels

Es handelt sich hierbei um eine initiale Belegung der Menge *selectedComponents* der einzelnen *Templates*. Anschließend werden die möglichen Variationen iterativ erzeugt. Hierbei können die Zeilen 5-15 des Konfigurationsalgorithmus, welcher in Listing 6-2 angegeben ist, durch die jeweilige selbstorganisierende Komponente durchgeführt werden, ohne dass der zentrale Konfigurator genaue Kenntnis über die möglichen Variationen besitzen muss. Jede Komponente erzeugt autonom eine lokale Konfiguration, welche zum einen die Anforderungen des jeweiligen *Templates* berücksichtigt, und zum anderen die komponentenspezifischen Anforderungen zur Ausführung der angebotenen Dienste erfüllt. Eine derartige Konfiguration ist in Abbildung 8-21 dargestellt.

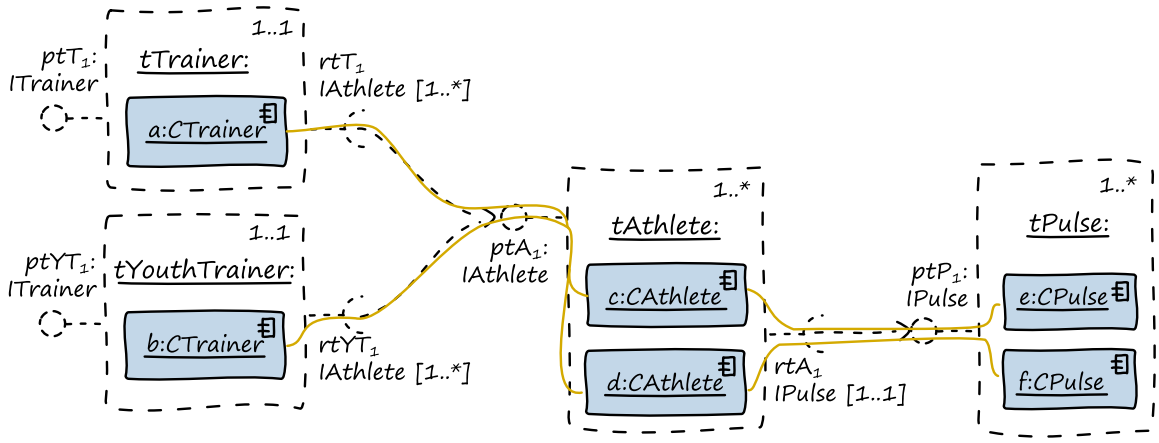


Abbildung 8-21: Eine initiale Konfiguration unter Berücksichtigung struktureller Anforderungen

Nachdem eine solche Konfiguration erzeugt wurde, wird das Prädikat *appConstraint* durch die zentrale Konfigurationseinheit ausgewertet. Wird es zu *true* ausgewertet, so wechselt die Anwendung in den Zustand RUNNING. Die einzelnen Komponenten können auf diesen Zustandswechsel beispielsweise damit reagieren, dass GUI-Komponenten die implementierte

Benutzungsschnittstelle anzeigen. Wird das Prädikat hingegen zu *false* ausgewertet, so muss eine neue Konfiguration erzeugt werden. Hierzu teilt die zentrale Konfigurationseinheit genau einer Komponente mit, dass die nächste Variation erzeugt werden soll. Für die Athletenkomponenten bieten sich hierbei insbesondere Variationen hinsichtlich der Auswahl an zu verwendenden Pulsdiensten zur Auflösung von Abhängigkeiten. So haben beide Athletenkomponenten die Wahl zwischen dem Pulsdienst der Komponente *e* und dem Dienst der Komponente *f*. Für die Trainerkomponenten bleibt in der dargestellten Situation die Wahl, beide Athletendienste zu verwenden, oder einen von beiden. Die Suche nach einer gültigen Konfiguration wird solange durchgeführt, bis der Vorgang durch Deinstallation der Anwendung gestoppt wird. Selbst, wenn bereits alle möglichen Variationen erzeugt und getestet wurden, wird mit dem Prozess fortgefahren, da jederzeit ein Zustandswechsel innerhalb einer Komponente stattfinden kann, der zu einer gültigen Konfiguration führen kann. Wurde eine gültige Konfiguration gefunden, wird zyklisch überprüft, ob diese Konfiguration noch die spezifizierten Anforderungen erfüllt.

8.6 Zusammenfassung

In diesem Kapitel wurde anhand einiger ausgewählter Szenarios in erster Linie die Funktionsweise der erarbeiteten Konzepte untersucht. Hierbei wurden die Szenarios derart ausgewählt, dass jeweils bestimmte Teilaspekte des Lösungskonzeptes beleuchtet werden konnten. Für den Fall, dass ein System ausschließlich aus selbstorganisierenden Komponenten zusammengesetzt ist, ohne Berücksichtigung komponentenübergreifender Anforderungen, wurden vier typische Szenarios beschrieben. Hierbei wurde zum einen gezeigt, wie die einzelnen Teilelemente einer Komponente während des Konfigurationsprozesses zusammenwirken, um eine lauffähige Konfiguration zu erzeugen. Zum anderen wurde evaluiert, wie die einzelnen Komponenten untereinander interagieren und bei Auftreten bestimmter Ereignisse reagieren. In allen Fällen ist das Ziel des Konfigurationsprozesses die Ausführung von Diensten mit gesetztem *requestRun*-Flag, unter der Berücksichtigung komponentenlokaler Prioritäten hinsichtlich zu aktivierender *ComponentConfigurations*. Im Rahmen der Evaluierung konnte gezeigt werden, dass dies durch die erarbeitete Lösung erreicht wird.

In einem abschließenden Szenario wurde in erster Linie die Ausdrucksmächtigkeit zur Spezifikation Anwendungsarchitektur-spezifischer diskutiert. Zudem wurde evaluiert, inwieweit der skizzierte Algorithmus zur Erzeugung einer Anwendungsarchitektur-spezifischen Konfiguration zum gewünschten Konfigurationsergebnis führt. Der Algorithmus verfolgt hierbei einen Brute-Force-Ansatz, bei dem im schlechtesten Fall sämtliche möglichen Konfigurationen erzeugt werden müssen, bevor eine gültige Konfiguration gefunden wurde. Auf Grund der Anzahl der Variationspunkte handelt es sich hierbei um einen Algorithmus mit exponentieller Laufzeitkomplexität. Jedoch war es auch nicht das Hauptziel der Arbeit, hierfür einen effizienten Algorithmus anzugeben. Auf Optimierungsmöglichkeiten wird zudem im letzten Kapitel dieser Arbeit noch ein wenig eingegangen. Hauptziel der Arbeit war es, überhaupt erst die Möglichkeit zu schaffen, in Systemen mit emergenten Eigenschaften Anwendungsarchitektur-spezifische Anforderungen definieren zu können. Dies bietet die hier vorgestellte Lösung, in dem zum einen Kriterien zur Auswahl von Komponenten definiert werden können, und zum anderen Kriterien für den Aufbau von Verbindungen zwischen den ausgewählten Komponenten.

9 Zusammenfassung und Ausblick

In diesem letzten Kapitel werden die einzelnen Teilaspekte dieser Arbeit noch einmal zusammengefasst und mit den definierten Zielen in Beziehung gesetzt. Abschließend wird ein Ausblick auf mögliche Ergänzungen und Verbesserungen der hier vorgestellten Konzepte gegeben.

9.1 Zusammenfassung

Die Entwicklung verteilter mobiler Anwendungen ist eine komplexe Aufgabe und erfordert die Berücksichtigung besonderer Anforderungen. Hierzu zählen insbesondere der Ausfall einzelner Komponenten, das Hinzukommen von Komponenten zur Laufzeit, sowie der Aufbau und der Abbruch von Kommunikationsverbindungen zwischen den Komponenten. Im Rahmen dieser Arbeit wurde ein Framework entwickelt, welches in der Lage ist, Entwicklern große Teile derjenigen Aufgaben abzunehmen, die der Realisierung dieser Anforderungen dienen.

Selbstorganisierende Komponenten als Ausgangspunkt

Die Entwicklung verteilter mobiler Anwendungen basierend auf selbstorganisierenden Komponenten birgt zahlreiche Vorteile. Insbesondere wird keine zentrale Konfigurationseinheit benötigt, welche sich um die Konfiguration des Gesamtsystems kümmern muss. Vielmehr konfiguriert sich jede Komponente autonom unter Berücksichtigung komponentenlokaler Anforderungen. Eine zentrale Konfigurationskomponente ist meist sehr komplex und ihre Entwicklung fehleranfällig. Zudem muss die Anwendung gestoppt werden, wenn diese Komponente ausfällt. Ein weiterer Nachteil zentral konfigurierter Systeme ist ein erhöhter Kommunikationsbedarf zwischen einzelnen Anwendungskomponenten und der zentralen Konfigurationseinheit. Insofern eignen sich selbstorganisierende Komponenten gut als Basis für die Entwicklung verteilter mobiler Anwendungen. Allerdings besitzt dieser Ansatz auch einige Nachteile. Der wesentliche Nachteil ist der, dass insbesondere komponentenübergreifende Anforderungen nicht ohne weiteres berücksichtigt werden können. Aus diesem Grunde wurde im Rahmen dieser Arbeit ein Ansatz vorgestellt, welcher die emergenten Eigenschaften selbstorganisierender Systeme auf Basis selbstorganisierender Komponenten erhält, aber gleichzeitig in der Lage ist, anwendungsspezifische, komponentenübergreifende Anforderungen automatisiert zur Laufzeit zu berücksichtigen.

Berücksichtigung Anwendungsarchitektur-spezifischer Anforderungen

In vielen Fällen entspricht ein System, welches derart konfiguriert wird, dass ausschließlich komponentenlokale Anforderungen berücksichtigt werden, nicht dem, was aus Anwendersicht optimal ist. So ist es in einigen Fällen sinnvoller, Anforderungen einzelner Komponenten gegenüber den Anforderungen an die Struktur des Gesamtsystems zurückzustellen. Das in dieser Arbeit vorgestellte Framework versetzt Anwendungsentwickler in die Lage, Anwendungsarchitektur-spezifische Anforderungen spezifizieren zu können und automatisiert umsetzen zu lassen, ohne die konkreten Komponenten zur Spezifikationszeit kennen zu müssen. Dadurch erlaubt das Konzept die Erhaltung des emergenten Charakters des Gesamtsystems.

Die Lösung basiert hierbei darauf, dass denjenigen Komponenten, welche an einer Anwendung beteiligt sind, Anforderungen durch die Anwendung induziert werden. Umgesetzt werden diese Anforderungen dann wiederum selbständig durch die Komponenten selbst. Allerdings wird in

diesem Fall eine zentrale Konfigurationseinheit benötigt, welche den Konfigurationsprozess steuert. Diese zentrale Einheit kann jedoch einfach gehalten werden, da sie lediglich steuert, welche Komponente die nächste Konfigurationsvariante erzeugen soll und zudem überprüft, ob die erzeugte Konfiguration den definierten, komponentenübergreifenden Anforderungen entspricht. Die zentrale Konfigurationskomponente benötigt allerdings, im Gegensatz zu zahlreichen anderen Ansätzen, kein Wissen über die genaue Systemstruktur, und muss die Abhängigkeiten zwischen Komponenten nicht selbst auflösen. Außerdem muss die Anwendung auch bei einem Ausfall der zentralen Komponente nicht zwingend gestoppt werden. Sie könnte weiterhin ausgeführt werden, dann allerdings ohne Berücksichtigung anwendungsspezifischer Anforderungen.

Implementierung

In Kapitel 7 wurde eine prototypische Implementierung der zuvor beschriebenen Konzepte vorgestellt. Die Implementierung besteht hierbei aus zwei Teilen. Zum einen wurde eine Implementierung beschrieben, welche die Entwicklung mobiler verteilter Anwendungen auf Basis selbstorganisierender Komponenten unter Berücksichtigung komponentenlokaler Anforderungen umsetzt. Dieser Teil der Implementierung realisiert somit die Konzepte aus den Kapiteln 4 und 5. Zum anderen wurde eine Implementierung skizziert, welche zudem die Umsetzung der in Kapitel 6 beschriebenen Konzepte erlaubt.

Evaluation

Im Rahmen dieser Arbeit wurde eine szenariobasierte Evaluation durchgeführt. Diese hat anhand des eingangs dieser Arbeit vorgestellten Anwendungsbeispiels die Vor- und Nachteile der erarbeiteten Lösungen untersucht und anhand der zuvor definierten Anforderungen bewertet. Hierbei wurde gezeigt, dass die entwickelten Konzepte die wesentlichen Aspekte berücksichtigt, die bei der Entwicklung mobiler verteilter Anwendungen basierend auf selbstorganisierenden Komponenten zu berücksichtigen sind. So unterstützt das Konzept zum einen die Entwicklung und Ausführung mobiler verteilter Anwendungen auf Basis selbstorganisierender Komponenten, wobei insbesondere die Berücksichtigung komponentenlokaler Anforderungen im Vordergrund steht. Zum anderen lassen sich mit Hilfe der Erweiterungen aus Kapitel 6 auf Basis eines Systems aus selbstorganisierenden Komponenten auch anwendungsspezifische, komponentenübergreifende Anforderungen spezifizieren und automatisiert umsetzen.

9.2 Diskussion und Ausblick

In diesem Abschnitt werden zum einen die erzielten Ergebnisse der vorliegenden Dissertation in Beziehung gesetzt zu den Zielen, welche in Abschnitt 1.2 formuliert wurden. Zum anderen werden hierzu jeweils mögliche Folgearbeiten, Schwächen und Ergänzungsmöglichkeiten diskutiert.

Eines der Ziele dieser Arbeit war es, die Entwicklung selbstorganisierender Systeme zu unterstützen, indem wesentliche Teile der Konfigurationslogik in ein Framework ausgelagert werden können (A1). Hierbei sollte der Entwickler allerdings weiterhin ausreichend Einfluss auf den Konfigurationsprozess nehmen können. Der vorgestellte Ansatz ermöglicht u.a. eine Unterscheidung zwischen Endanwenderdiensten und Diensten, welche vorrangig durch andere Komponenten im System verwendet werden. Dies wird durch das Flag *requestRun* erreicht. Des

Weiteren können für eine Komponente Anforderungen hinsichtlich der Auflösung von Abhängigkeiten von anderen Komponenten definiert werden (A2). Hierzu zählen insbesondere die Angabe eines Intervalls sowie die Berücksichtigung von Dienstzuständen mit Hilfe von Schnittstellenrollen. Die erarbeiteten Konzepte wurden zudem in Form eines Frameworks prototypisch umgesetzt und getestet.

Die Möglichkeiten zur Einflussnahme auf den Konfigurationsprozess können allerdings noch weiter ausgebaut werden. So sieht der Ansatz momentan nicht die Möglichkeit vor, innerhalb eines *RequiredServiceReferenceSets* eine Ordnungsrelation über zur Verfügung stehende Dienste definieren zu können. Soll eine Komponente beispielsweise nur mit denjenigen Diensten verbunden sind, die der Komponente örtlich am nächsten sind, so lässt sich das mit dem hier vorliegenden Ansatz noch nicht definieren. Die Angabe einer maximalen Entfernung ist hingegen mit Hilfe von Schnittstellenrollen wiederum möglich.

Weitere Anknüpfungspunkte für zukünftige Arbeiten bietet der angegebene dezentrale Konfigurationsalgorithmus für selbstorganisierende Komponenten. So erfolgt eine Dienstzuteilung aktuell nach dem Prinzip *first-come-first-serve*. An dieser Stelle wäre es u.a. denkbar, bei der Zuteilung von Diensten an *RequiredServiceReferenceSets* bestimmte Prioritäten zu berücksichtigen, sowie die Möglichkeit zu geben, Dienste einem Nutzer wieder entziehen zu können. Im vorgelegten Konzept behält ein *RequiredServiceReferenceSet* solange die Nutzungsrechte, bis er den Dienst wieder freigibt, oder bis der Dienst beendet wird.

An diesem Punkt ergeben sich weitere Optimierungsmöglichkeiten. Wird ein Dienst beendet, führt das ggf. dazu, dass eine oder mehrere hiervon abhängige Komponenten nicht mehr ausgeführt werden können. In der vorliegenden Lösung werden abhängige Komponenten unmittelbar angehalten, wenn sie keine aktivierbare Komponentenkonfiguration mehr besitzen. In einigen Fällen wäre es hier allerdings wünschenswert, zunächst eine kurze Zeit abzuwarten, ob ein geeigneter Ersatz zur Verfügung steht, um so ggf. umfangreiche Rekonfigurationen bei kurzen Dienstausfällen verhindern zu können.

Gerade in mobilen verteilten Systemen kann es vorkommen, dass Verbindungen zwischen Komponenten unterbrochen werden, oder Komponenten unvorhergesehen ausfallen. Das vorgestellte Konzept sieht vor, dass sich eine Komponente durch Aufruf von *uninstall* aus dem System entfernt. Bei einem unvorhergesehenen Ausfall einer Verbindung oder einer Komponente wäre ein solcher Aufruf nicht mehr möglich. Eine gängige, und einfach einzuarbeitende Lösung wäre die Einführung eines Watchdog-Konzeptes, wie es u.a. die Middelware Jini [Edw00] anbietet. Auf Seiten des Dienstverwenders wird hierzu ein Thread gestartet, welcher die Verfügbarkeit verwendeter Dienste zyklisch überprüft und bei unerwartetem Wegfall einer Komponente die nötigen Konfigurationsmaßnahmen initiiert.

Auch bei der Umsetzung des Schnittstellenrollenkonzeptes gibt es einige Erweiterungsmöglichkeiten. Ein wesentlicher Aspekt hierbei ist die durchgängige Sicherstellung der Rollenkonformität von Diensten, bzw. unmittelbar zu reagieren, sollte die Konformität zu einem Zeitpunkt nicht mehr gegeben sein. In dieser Arbeit wurde eine Lösung vorgestellt, welche auf der zyklischen Überprüfung der Rollenzugehörigkeit basiert. Zwischen zwei Überprüfungszeitpunkten kann eine Rollenkonformität allerdings nicht gewährleistet werden. Ein testbasiertes System, welches in [Nie10] zur Überprüfung von Schnittstellenkonformität zur Laufzeit vorgestellt wurde, könnte hierzu in den hier vorgestellten Ansatz integriert werden.

In Kapitel 6 wurde ein Konzept vorgestellt, welches zunächst die Möglichkeit bietet, Anwendungsarchitektur-spezifische Anforderungen spezifizieren zu können (A3). Hierbei kann ein Anwendungsentwickler Bedingungen festlegen, unter denen eine Komponente in die Anwendung aufgenommen werden soll, ohne die konkreten Komponententypen und Komponenteninstanzen kennen zu müssen. Zusätzlich bietet der Ansatz die Möglichkeit, Vorgaben hinsichtlich der Vernetzung der Komponenten machen zu können. Neben den Möglichkeiten der Spezifikation Anwendungsarchitektur-spezifischer Anforderungen wurde in dieser Arbeit zudem ein Ansatz zur automatisierten Erzeugung einer anforderungskonformen Konfiguration vorgestellt, sowie ein Implementierungskonzept.

Auch hier ergeben sich weitere Anknüpfungspunkte für zukünftige Arbeiten. So lassen sich zwar Kriterien definieren, unter denen eine Komponente in eine Anwendung aufgenommen werden soll, jedoch sieht das Konzept bislang nicht vor, eine Ordnung über diese Komponenten spezifizieren zu können. So kann mit den eingeführten Mitteln beispielsweise nicht spezifiziert werden, dass für ein *Template* nur diejenigen fünf Komponenten verwendet werden sollen, die dem Anwender örtlich am nächsten sind.

Optimierungspotential bietet zudem der angegebene Konfigurationsalgorithmus zur Erzeugung einer Anwendungsarchitektur-konformen Konfiguration. Dieser basiert zurzeit darauf, sämtliche möglichen Konfigurationen nacheinander zu erzeugen und auf Konformität zu überprüfen. Neben zahlreichen anderen Optimierungsmöglichkeiten, bietet eine Integration von *Event-Condition-Action-Rules* in den Ansatz eine vielversprechende. Diese ECA-Regeln müssten durch den Anwendungsentwickler definiert werden, und beschreiben durchzuführende Rekonfigationsmaßnahmen beim Auftreten bestimmter, häufig auftretender Ereignisse. Auf diese Weise könnte die Rekonfiguration für häufig auftretende Fälle effizient durchgeführt werden, ohne im ungünstigsten Fall sämtliche Konfigurationsvarianten erzeugen und testen zu müssen.

Im Rahmen dieser Dissertation wurde ein Framework entwickelt, welches die vorgestellten Konzepte realisiert und so den Entwicklern die Möglichkeit bietet, sich auf die Realisierung der fachlichen Logik konzentrieren zu können. Durch den Einsatz grafischer Modellierungswerkzeuge sowohl für Anwendungen als auch für Komponenten könnten diese Möglichkeiten noch weiter verbessert werden. So wäre es denkbar, dass Komponentenentwicklern ein Werkzeug bereitgestellt wird, in dem der die einzelnen benötigten Elemente seiner Komponente (*ProvidedServices, ComponentConfigurations*, …) per Drag&Drop zusammenfügt. Das Werkzeug könnte basierend darauf dann bereits den Code für das Grundgerüst der Komponente generieren. Ähnliche Lösungen sind für die Unterstützung der Anwendungsentwicklung vorstellbar, um die Erstellung von Anwendungsspezifikationen zu vereinfachen.

Literaturverzeichnis

[Abt07] ABTS, Dietmar: *Masterkurs Client/Server-Programmierung mit Java.* 2. Aufl.: Vieweg, 2007. – ISBN 3834803227

[ACG+05] AGRAWAL, Dakshi; CALO, Seraphin; GILES, James; LEE, Kang-Won; VERMA, Dinesh: Policy Management for Networked Systems and Applications. In: CLEMM, Alexander; FESTOR, Olivier; PRAS, Aiko (Hrsg.): *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management.* 15-19 May 2005, Nice, France, 2005, S. 455–468

[AEP+07] ALIA, Mourad; EIDE, Victor S. Wold; PASPALLIS, Nearchos; ELIASSEN, Frank; HALLSTEINSEN, Svein O.; PAPADOPOULOS, George A.: A Utility-based Adaptivity Model for Mobile Applications. In: *Proceedings of the 21st International Conference on Advanced Information Networking and Applications.* Niagara Falls, Ontario, Canada, 21-23 May 2007. Los Alamitos, California: IEEE Computer Society Press, 2007, S. 556–563

[AHE+06] ALIA, Mourad; HORN, Geir; ELIASSEN, Frank; KHAN, Mohammad Ullah; FRICKE, Rolf; REICHLE, Roland: A Component-Based Planning Framework for Adaptive Systems. In: MEERSMANN, Robert; TARI, Zahir (Hrsg.): *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE.* Montpellier, France, October 29 - November 3, 2006. Berlin Heidelberg: Springer, 2006 (Lecture Notes in Computer Science, 4276), S. 1686–1704

[AkC03] AKSIT, Mehmet; CHOUKAIR, Zièd: Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshop.* 19-22 May 2003, Providence, Rhode Island, USA. Los Alamitos, California : IEEE Computer Society Press, 2003, S. 84–89

[Ald03] ALDRICH, Jonathan: *Using Types to Enforce Architectural Structure.* Washington, DC, USA, University of Washington, Computer Science and Engineering. Dissertation. 2003

[AsZ12] ASCHOFF, Rafael R.; ZISMAN, Andrea: Proactive adaptation of service composition. In: MÜLLER, Hausi A.; BARESI, Luciano (Hrsg.): *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12): Zürich, Switzerland, June 4-5, 2012.* Los Alamitos, California: IEEE Computer Society Press, 2012, S. 1–10

[Bal00] BALZERT, Helmut; BALZERT, Heide (Mitarb.); KOSCHKE, Rainer (Mitarb.); LÄMMEL, Uwe (Mitarb.); LIGGESMEYER, Peter (Mitarb.); QUANTE, Jochen (Mitarb.): *Lehrbuch der Software-Technik.* 3. Aufl. : Spektrum Akademischer Verlag, 2000

[BBC+02] BATRA, Vishal S.; BHATTACHARAYA, Jaijit; CHAUHAN, Harish; GUPTA, Ajay; MOHANIA, Mukesh; SHARMA, Upendra: Policy Driven Data Administration. In: *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks.* June 5-7, 2002, Monterey, California, USA. Los Alamitos, California : IEEE Computer Society Press, 2002, S. 220–223

[BBG+08] BRUNI, Roberto; BUCCIARONE, Antonio; GNESI, Stefania; MELGRATTI, Hernán: *Modelling Dynamic Software Architectures using Typed Graph Grammars.* In: *Electronic Notes in Theoretical Computer Science* 213 (2008), Nr. 1, S. 39–53

[BDH+98] BROY, Manfred; DEIMEL, Anton; HENN, Jürgen; KOSKIMIES, Kai; PLASIL, Frantisek; POMBERGER, Gustav; PREE, Wolfgang; STAL, Michael; SZYPERSKI, Clemens A.: *What characterizes a (software) component?* In: *Software - Concepts and Tools* 19 (1998), Nr. 1, S. 49–56

[BNM+08] BINDELLI, Silvia; NITTO, Elisabetta Di; MIRANDOLA, Raffaela; TEDESCO, Roberto: Building Autonomic Components: the SelfLets Approach. In: *Proceedings of the 1st International Workshop on Automated Engineering of Autonomous and Runtime Evolving Systems.* September 16, 2008 L'Aquila, Italy, 2008, S. 17–24

[Bos04] BOSSEL, Hartmut: *Systeme, Dynamic, Simulation: Modellbildung, Analyse und Simulation komplexer Systeme.* Norderstedt, Germany: Books on Demand Gmbh, 2004. – ISBN 3833409843

[Bun79] BUNGE, Mario: *A World of Systems.* Dordrecht, Holland: Reidel, 1979 (Treatise on Basic Philosophy 4)

[CCS09] CÁMARA, Javier; CANAL, Carlos; SALAÜN, Gwen: Behavioural Self-Adaptation of Services in Ubiquitous Computing Environments. In: *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems.* May 18-19, 2009, Vancouver, BC, Canada: IEEE Computer Society Press, 2009, S. 28–37

[CFB01] CUESTA, Carlos E.; FUENTE, Pablo de la; BARRIO-SOLÁRZANO, Manuel: Dynamic Coordination Architecture through the use of Reflection. In: LAMONT, Gary B. (Hrsg.): *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC'01).* Las Vegas, Nevada, United States. New York, NY, USA: ACM-Press, 2001, S. 134–140

[CGS06] CHENG, Shang-Wen; GARLAN, David; SCHMERL, Gradley: Architecture-based Self-Adaptation in the Presence of Multiple Objectives. In: CHENG, Betty H.C.; LEMOS, Rogério de; FICKAS, Stephan; GARLAN, David; MAGEE, Jeff; MÜLLER, Hausi; TAYLOR, Richard (Hrsg.): *Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems.* May 21-22, 2006, Shanghai, China. New York, NY, USA: ACM-Press, 2006, S. 2–8

[ChD00] CHEESMAN, John; DANIELS, John: *UML Components: A Simple Process for Specifying Component-Based Software:* Addison-Wesley, 2000

[Che08] CHENG, Shang-Wen: *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation.* Pittsburgh, Carnegie Mellon University, School of Computer Science. Dissertation. 2008

[CHS02] CHEN, Wen-Ke; HILTUNEN, Matti A.; SCHLICHTING, Richard D.: Constructing Adaptive Software in Distributed Systems. In: *Proceedings of the 21st International Conference on Distributed Computing Systems.* ICDCS 2001, April 16-19, 2001, Phoenix, Arizona, USA : IEEE Computer Society Press, 2001, S. 635–643

[CPC08] COSTA-SORIA, Cristóbal; PÉREZ, Jennifer; CARSÍ, José Angel: Managing Dynamic Evolution of Architectural Types. In: MORRISON, R.; BALASUBRAMANIAM, D.; FALKNER, K. (Hrsg.): *Software Architecture.* Berlin Heidelberg: Springer, 2008 (Lecture Notes in Computer Science, 5292), S. 281–289

[CSW+04] CHESS, David M.; SEGAL, Alla; WHALLEY, Ian; WHITE, Steve R.: Unity: Experiences with a Prototype Autonomic Computing System. In: *Proceedings of the 1st International Conference on Autonomic Computing.* 17-19 May 2004, New York, NY, USA. Washington, DC, USA: IEEE Computer Society Press, 2004, S. 140–147

[DDL+00] DAMIANOU, Nicodemos; DULAY, Naranker; LUPU, Emil; SLOMAN, Morris: *Ponder: A Language for Specifying Security and Management Policies for Distributed Systems.* 2000(Imperial College Research Report DoC 2000/1)

[DeP00] DEVAUL, Richard W.; PENTLAND, Alex: *The Ektara Architecture: The Right Framework for Context-Aware Wearable and Ubiquitous Computing Applications.* 2000

[Dij72] DIJKSTRA, Edsger W.: *The Humble Programmer.* In: *Communications of the ACM* 15 (1972), Nr. 10, S. 859–866

[DJM+04] DESMET, Lieven; JANSSENS, Nico; MICHIELS, Sam; PIESSENS, Frank; JOOSEN, Wouter; VERBAETEN, Pierre: Towards Preserving Correctness in Self-Managed Software Systems. In: GARLAN, David; KRAMER, Jeff; WOLF, Alexander (Hrsg.): *Proceedings of the 1st International Workshop on Self-Managed Systems (WOSS '04).* Newport Beach, CA, USA, October 31-November 01. New York, NY, USA: ACM-Press, 2004, S. 34–38

[Dob12] DOBRIC, Damir: *Managed Extensibility Framework.* URL http://msdn.microsoft.com/de-de/library/ee332203.aspx

[Edw00] EDWARDS, W. Keith: *Core Jini.* 2. Aufl.: Prentice Hall International, 2000

[FKS97] FICKAS, Stephen; KORTUEM, Gerd; SEGALL, Zary: Software Organisation for Dynamic and Adaptable Wearable Systems. In: *Proceedings of the 1st International Symposium on Wearable Computers.* October 13-14, 1997, Cambridge, Massachusetts. Los Alamitos, California : IEEE Computer Society Press, S. 56–63

[GaC03] GANEK, A. G.; CORBI, T. A.: *The dawning of the autonomic computing era.* In: *IBM Systems Journal* 42 (2003), Nr. 1, S. 5–18

[GaP95] GARLAN, David; PERRY, Dewayne E.: *Introduction to the Special Issue on Software Architecture.* In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 4, S. 269–274

[GBS03] GRACE, Paul; BLAIR, Gordon S.; SAMUEL, Sam: ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. In: MEERSMAN, Robert; TARI, Zahir; SCHMIDT, Douglas C. (Hrsg.): *Proceedings of the International Conference on Distributed Objects and Applications.* Catania, Siciliy, Italy, November 3-7, 2003: Springer, 2003 (Lecture Notes in Computer Science, 2888), S. 1170–1187

[GCH+04] GARLAN, David; CHENG, Shang-Wen; HUANG, An-Cheng; SCHMERL, Bradley; STEENKISTE, Peter: *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure.* In: *Computer* 37 (2004), Nr. 10, S. 46–54

[GHJ94] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph E.: *Design Patterns: Elements of Reusable Object-Oriented Software:* Addison-Wesley Longman, 1994

[GMK02] GEORGIADIS, Ioannis; MAGEE, Jeff; KRAMER, Jeff: Self-Organising Software Architectures for Distributed Systems. In: GARLAN, David; KRAMER, Jeff; WOLF, Alexander L. (Hrsg.): *Proceedings of the First Workshop on Self-Healing Systems.* WOSS 2002, Charleston, South Carolina, USA, November 18-19, 2002: ACM-Press, 2002. – ISBN 1-58113-609-9, S. 33–38

[GMW00] GARLAN, David; MONROE, Robert T.; WILE, David: Acme: architectural description of component-based systems. In: LEAVENS, Gary T.; SITARAMAN, Murali (Hrsg.): *Foundations of component-based systems :* Cambridge University Press New York, NY, USA, 2000. – ISBN 0-521-77164-1, S. 47–67

[GrT00] GRUHN, Volker; THIEL, Andreas: *Komponentenmodelle: DCOM, JavaBeans, Enterprise JavaBeans, CORBA.* München: Addison-Wesley, 2000 (Professionelle Softwareentwicklung). – ISBN 382731724X

[HeC01] HEINEMANN, George T.; COUNCILL, William T.: *Component-Based Software Engineering: Putting the Pieces Together.* Amsterdam: Addison-Wesley Longman, 2001. – ISBN 0201704854

[HiS96] HILTUNEN, Matti A.; SCHLICHTING, Richard D.: *Adaptive Distributed and Fault-Tolerant Systems.* In: *International Journal of Computer Systems Science and Engineering* 11 (1996), Nr. 5, S. 275–285

[HKL+84] HESSE, W.; KEUTGEN, H.; LUFT, A. L. ; ROMBACH, D.: *Ein Begriffssystem für die Softwaretechnik - Vorschlag zur Terminologie.* In: *Informatik-Spektrum* (1984), Nr. 7, S. 200–213

[HnP06] HNETYNKA, Petr; PLÁSIL, Frantisek: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models. In: GORTON, Ian; HEINEMANN, George T.; CRNKOVIC, Ivica; SCHMIDT, Heinz W.; STAFFORD, Judith A.; SZYPERSKI, Clemens; WALLNAU, Kurt (Hrsg.): *Component-Based Software Engineering (CBSE '06).* Berlin Heidelberg: Springer, 2006 (Lecture Notes in Computer Science, 4063), S. 352–359

[Hor01] HORN, Paul: *Autonomic Computing: IBM's Perspective on the State of Information Technology.* URL http://www.research.ibm.com/autonomic/manifesto/

[HPM+05] HNETYNKA, Petr; PLÁSIL, Frantisek; MENCL, Vladimír; KAPOVÁ, Lucia: *SOFA 2.0 metamodel.* 2005(11/2005)

[HSM+09] HEAVEN, William; SYKES, Daniel; MAGEE, Jeff; KRAMER, Jeff: A Case Study in Goal-Driven Architectural Adaptation. In: CHENG, Betty H.C.; LEMOS, Rogério de; GIESE, Holger; INVERARDI, Paola; MAGEE, Jeff (Hrsg.): *Software Engineering for Self-Adaptive Systems.* Berlin Heidelberg: Springer, 2009 (Lecture Notes in Computer Science, 5525), S. 109–127

[Ibm05] IBM: *An architectural blueprint for autonomic computing.* 3. Aufl. 2005

[KBW+94] KAZMAN, Rick; BASS, Len; ABOWD, Gregory; WEBB, Mike: SAAM: A Method for Analyzing the Properties of Software Architectures. In: FADINI, Bruno; OSTERWEIL, Leon; LAMSWEERDE, Axel van (Hrsg.): *Proceedings of the 16th International Conference on Software Engineering.* May 16-21, 1994, Sorrento, Italy. Los Alamitos, California, USA : IEEE Computer Society Press, 1994, S. 81–90

[KeC03] KEPHART, Jeffrey O.; CHESS, David M.: *The Vision of Autonomic Computing.* In: *Computer* 36 (2003), Nr. 1, S. 41–50

[KKB+98] KAZMAN, Rick; KLEIN, Mark; BARBACCI, Mario; LONGSTAFF, Tom; LIPSON, Howard; CARRIERE, Jeromy: *The Architecture Tradeoff Analysis Method.* 1998(CMU/SEI-98-TR-008)

[KMS+92] KRAMER, Jeff; MAGEE, Jeff; SLOMAN, Morris; DULAY, Naranker: *Configuring Object-Based Distributed Programs in REX.* In: *Software Engineering Journal* 7 (1992), Nr. 2, S. 139–149

[Kos00] KOSCH, Andreas: *COM/DCOM/COM+ mit Delphi.* 2. Aufl.: Software & Support, 2000

[Kra90] KRAMER, Jeff: Configuration Programming: A Framework for the Development of Distributable Systems. In: *Proceedings of IEEE International Conference on Computer Systems and Software Engineering (COMPEURO 90).* 8-10 May 1990, Tel-Aviv, Israel: IEEE Computer Society Press, 1990. – ISBN 0818620412, S. 374–384

[KrM85] KRAMER, Jeff; MAGEE, Jeff: *Dynamic Configuration for Distributed Systems.* In: *IEEE Transactions on Software Engineering* 11 (1985), Nr. 4, S. 424–436

[KrR08] KROGMANN, Klaus; REUSSNER, Ralf: Palladio - Prediction of Performance Properties. In: RAUSCH, Andreas; REUSSNER, Ralf; MIRANDOLA, Raffaela; PLASIL, Frantisek (Hrsg.): *The Common Component Modeling Example: Comparing Software Component Models :* Springer, 2008 (Lecture Notes in Computer Science, 5153). – ISBN 978-3-540-85288-9, S. 297–326

[Lad97] LADDAGA, Robert: *DARPA Broad Agency Announcement on Self Adaptive Software.* 1997(BAA 98-12)

[LaT09] LALEHIE, Mazeiar; TAHVILDARI, Ladan: *Self-adaptive software: Landscape and research challenges.* In: *ACM Transactions on Autonomous and Adaptive Systems* 4 (2009), Nr. 2, S. 14:1–14:42

[LBN99] LOBO, Jorge; BHATIA, Randeep; NAGVI, Shamim: A Policy Description Language. In: *Proceedings of the 16th National Conference on Artificial Intelligence and the 11th Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence : Menlo Part, CA, USA,* 1999, S. 291–298

[LGS+06] LUO, Weiping; GHENNIWA, Hamada; SHEN, Weiming; RAO, Zhaohua: Capture Dynamic Aspects of Software Architecture for Distributed Self-Adaptive Environments. In: LIN, Zongkai; BARTHÈS, Jean-Paul; SHEN, Weiming (Hrsg.): *Proceedings of the 10th International Conference on Computer Supported Cooperative Work in Design (CSCWD'06).* Nanjing, China: IEEE Computer Society Press, 2006, S. 544–549

[LLS03] LYMBEROPOULOS, Leonidas; LUPU, Emil; SLOMAN, Morris: *An Adaptive Policy-Based Framework for Network Services Management.* In: *Journal of Network and Systems Management* 11 (2003), Nr. 3, S. 277–303

[LMK+01] LUTFIYYA, Hanan; MOLENKAMP, Gary; KATCHABAW, Michael; BAUER, Michael A.: Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework. In: SLOMAN, Morris; LOBO, Jorge; LUPU, Emil (Hrsg.): *Proceedings of the International Workshop on Policies for Distributed Systems and Networks.* 29-31 January 2001, Bristol, UK. London, UK : Springer, 2001 (Lecture Notes in Computer Science, 1995), S. 185–201

[LuS97] LUPO, Emil; SLOMAN, Morris: Conflict Analysis for Management Policies. In: LAZAR, Aurel A.; SARACCO, Roberto; STADLER, Rolf (Hrsg.): *Proceedings of the 5th International Symposium on Integrated Network Management.* May 12-16, 1997, San-Diego, CA, USA : Chapman & Hall, Ltd., 1997 (IFIP Conference Proceedings, 86), S. 430–443

[MaF10] MATTERN, Friedmann ; FLÖRKEMEIER, Christian: *Vom Internet der Computer zum Internet der Dinge.* In: *Informatik-Spektrum* 33 (2010), Nr. 2, S. 107–121

[MaK96] MAGEE, Jeff; KRAMER, Jeff: Self Organising Software Architectures. In: *Joint Proceedings of the 2nd International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96).* New York: ACM-Press, 1996, S. 35–38

[Mak12] MAKEWAVE AB: *Knopflerfish - Open Source OSGi.* URL http://www.knopflerfish.org/

[Mel10] MELZER, Ingo: *Service-orientierte Architekturen mit Web Servicees: Konzepte - Standards - Praxis.* 4. Aufl. : Spektrum Akademischer Verlag, 2010

[MKS89] MAGEE, Jeff; KRAMER, Jeff; SLOMAN, Morris: *Constructing Distributed Systems in Conic.* In: *IEEE Transactions on Software Engineering* 15 (1989), Nr. 6, S. 663–675

[Moo65] MOORE, Gordon E.: *Cramming More Components onto Integrated Circuits.* In: *Electronics* 38 (1965), Nr. 8, S. 114–117

[MRP+07] MICHLMAYR, Anton; ROSENBERG, Florian; PLATZER, Christian; TREIBER, Martin; DUSTDAR, Schahram: Towards Recovering the Broken SOA Triangle - A Software Engineering Perspective. In: NITTO, Elisabetta Di; POLINI, Andrea; ZISMAN, Andrea (Hrsg.): *Proceedings of the 2nd International Workshop on Service oriented software engineering (IW-SOSWE'07).* Dubrovnik, Croatia. New York, NY, USA: ACM-Press, 2007, S. 22–28

[MSK+04] MCKINLEY, Philip K.; SADJADI, Seyed Masoud; KASTEN, Eric P.; CHENG, Betty H.C.: *A Taxonomy of Compositional Adaptation.* Technical Report. 2004 (MSU-CSE-04-17). – Technical Report

[NaR68] NAUR, Peter (Hrsg.); RANDELL, Brian (Hrsg.): *Software Engineering: Report of a conference sponsored by the NATO Science Committee.* Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, 1969

[NGM+08] NITTO, Elisabetta Di; GHEZZI, Carlo; METZGER, Andreas; PAPAZOGLOU, Mike; POHL, Klaus: *A journey to highly dynamic, self-adaptive service-based applications.* In: *Automated Software Engineering* 15 (2008), 3-4, S. 313–341

[Nie10] NIEBUHR, Dirk: *Dependable Dynamic Adaptive Systems: Approach, Model, and Infrastructure.* Clausthal-Zellerfeld, Technische Universität Clausthal, Institut für Informatik. Dissertation. 2010

[NKA+07] NIEBUHR, Dirk; KLUS, Holger; ANASTASOPOULOS, Michalis; KOCH, Jan; WEIß, Oliver; RAUSCH, Andreas: *DAiSI : Dynamic Adaptive System Infrastructure.* 2007(IESE Report 051.07/E)

[Obj01] OBJECT MANAGEMENT GROUP: *Common Object Request Broker Architecture (CORBA) Specification, Version 3.2.* URL http://www.omg.org/spec/CORBA/3.2/ – Überprüfungsdatum 2012-02-23

[Obj06] OBJECT MANAGEMENT GROUP: *Object Constraint Language.* Version 2.0. URL http://www.omg.org/spec/OCL/2.0/. – Aktualisierungsdatum: 2012-05-29

[Obj09] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language Superstructure.* Version 2.2. URL http://www.omg.org/spec/UML/2.2/Superstructure. – Aktualisierungsdatum: 2009 – Überprüfungsdatum 2012-02-28

[Ore96] OREIZY, Peyman: *Issues in the Runtime Modification of Software Architectures.* 1996(UCI-ICS-TR-96-35)

[Osg11] OSGI ALLIANCE: *OSGi Service Platform Core Specification.* URL http://www.osgi.org/Specifications/HomePage – Überprüfungsdatum 2011-02-23

[PAC06] PÉREZ, Jennifer; ALI, Nour; CARSÍ, Jose A.; RAMOS, Isidro: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In: GORTON, Ian; HEINEMAN, George T.; CRNKOVIC, Ivica; SCHMIDT, Heinz W.; STAFFORD, Judith A.; SZYPERSKI, Clemens; WALLNAU, Kurt C. (Hrsg.): *Component-Based Software Engineering :* Springer, 2006 (Lecture Notes in Computer Science, 4063), S. 123–138

[Pap03] PAPAZOGLOU, Mike P.: Service-Oriented Computing: Concepts, Characteristics and Directions. In: *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003).* 10-12 December, Rome, Italy: IEEE Computer Society Press, 2003, S. 3–12

[Par72] PARNAS, David Lorge: *On the Criteria To Be Used in Decomposing Systems into Modules.* In: *Communications of the ACM* 15 (1972), Nr. 12, S. 1053–1058

[Pat11] PATERÒ, Fabio (Hrsg.): *Migratory Interactive Applications for Ubiquitous Environments :* Springer, 2011 (Human-Computer Interaction Series)

[PCH+12] PATIKIRIKORALA, Tharindu; COLMAN, Alan; HAN, Jun; WANG, Liupeng: A Systematic Survey on the Design of Self-Adaptive Software Systems Using Control Engineering Approaches. In: MÜLLER, Hausi A.; BARESI, Luciano (Hrsg.): *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12): Zürich, Switzerland, June 4-5, 2012.* Los Alamitos, California: IEEE Computer Society Press, 2012, S. 33–42

[PYP02] PONNAPPAN, Appan; YANG, Lingjia; PILLAI, Radhakrishna R.: A Policy Based QoS Management System for the IntServ/DiffServ Based Internet. In: *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks.* June 5-7, 2002, Monterey, California, USA. Los Alamitos, California : IEEE Computer Society Press, 2002, S. 159–168

[RaP02] RASCHE, Andreas; POLZE, Andreas: Configurable Services for Mobile Users. In: *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time*

Dependable Systems (WORDS 2002). 7-9 January 2002, San Diego,. Los Alamitos, California: IEEE Computer Society Press, 2002. – ISBN 0-7695-1576-2, S. 163–170

[RaP03] RASCHE, Andreas; POLZE, Andreas: Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET. In: *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003).* 14-16 May 2003, Hakodate, Hokkaido, Japan: IEEE Computer Society Press, 2003. – ISBN 0-7695-1928-8, S. 164–171

[Rau01] RAUSCH, Andreas: *Componentware: Methodik des evolutionären Architekturentwurfs.* München, Technische Universität München. Dissertation. 2001

[Rau07] RAUSCH, Andreas: *DisCComp - A Formal Model for Distributed Concurrent Components.* In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 176 (2007), Nr. 2, S. 5–23

[ReA07] RELLERMEYER, Jan S.; ALONSO, Gustavo: Concierge: A Service Platform for Resource-Constrained Devices. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems.* New York, NY, USA: ACM-Press, 2007, S. 245–258

[RJS+06] RAZ, Danny; JUHOLA, Arto; SERRAT-FERNANDEZ, Joan; GALIS, Alex: *Fast and Efficient Context-Aware Services :* Wiley, 2006

[RRM+08] RAUSCH, Andreas (Hrsg.); REUSSNER, Ralf (Hrsg.); MIRANDOLA, Raffaela (Hrsg.): *The Common Component Modeling Example: Comparing Software Component Models:* Springer, 2008 (Lecture Notes in Computer Science 5153). – ISBN 978-3-540-85288-9

[RuB10] RUBINGER, Andrew Lee; BURKE, Bill: *Enterprise JavaBeans 3.1.* 6. Aufl. : O'Reilly Media, 2010. – ISBN 0596158025

[SAW94] SCHILIT, Bill N.; ADAMS, Norman; WANT, Roy: Context-Aware Computing Applications. In: CABRERA, Luis-Felipe; SATYANARAYANAN, Mahadev (Hrsg.): *Proceedings of the Workshop on Mobile Computing Systems and Applications.* December 8-9, 1994, Santa Cruz, California. Los Alamitos, California: IEEE Computer Society Press, 1994, S. 85–90

[Sch93] SCHILIT, Bill N.: Customizing Mobile Applications. In: *Proceedings of the USENIX Symposium on Mobile and Location-independent Computing.* Cambridge, MA, 1993, S. 129–138

[SDA99] SALBER, Daniel; DEY, Anind K.; ABOWD, Gegory D.: The Context Toolkit: Aiding the Development of Context-Enabled Applications. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* May 15-20, 1999, Pittsburgh, Pennsylvania, United States. New York, NY, USA : ACM-Press, 1999, S. 434–441

[SoG02] SOUSA, Joao Pedro; GARLAN, David: Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In: BOSCH, Jan; GENTLEMAN, Morven W.; HOFMEISTER, Christine; KUUSELA, Juha (Hrsg.): *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Archtiecture.* August 25-30, 2002, Montréal, Québec, Canada, 2002 (IFIP Conference Proceedings, 224), S. 29–43

[SoW98] D'SOUZA, Desmond Francis; WILLS, Alan Cameron: *Objects, Components, and Frameworks with UML: The Catalysis Approach :* Addison-Wesley, 1998

[SRG+08] SCHÄFER, Jan; REITZ, Markus; GAILLOURDET, Jean-Marie; POETZSCH-HEFFTER, Arnd: Linking Programs to Architectures: An Object-Oriented Hierarchical Software Model Based on Boxes. In: RAUSCH, Andreas; REUSSNER, Ralf; MIRANDOLA, Raffaela; PLASIL, Frantisek (Hrsg.): *The Common Component Modeling Example: Comparing Software*

Component Models : Springer, 2008 (Lecture Notes in Computer Science, 5153). – ISBN 978-3-540-85288-9, S. 239–266

[Szy98] SZYPERSKI, Clemens: *Component Software: Beyond object-oriented programming.* New York: ACM-Press, 1998 (ACM Press books). – ISBN 0-201-17888-5

[Szy02] SZYPERSKI, Clemens; GRUNTZ, Dominik (Mitarb.); MURER, Stephan (Mitarb.): *Component Software: Beyond Object-Oriented Programming.* 2. Aufl. New York: Addison-Wesley, 2002

[Ull11] ULLENBOOM, Christian: *Java ist auch eine Insel: Das Umfassende Handbuch.* 10. Aufl. : Galileo Computing, 2011. – ISBN 383621802X

[W3c04] W3C: *Web Services Architecture.* URL http://www.w3.org/TR/ws-arch/

[WaS95] WARREN, Ian; SOMMERVILLE, Ian: Dynamic Configuration Abstraction. In: SCHÄFER, Wilhelm; BOTELLA, Pere (Hrsg.): *Proceedings of the 5th European Software Engineering Conference (ESEC '95).* Sitges, Spain, September 25-28, 1995: Springer, 1995 (Lecture Notes in Computer Science, 989). – ISBN 3-540-60406-5, S. 173–190

[Wei91] WEISER, Mark: *The Computer for the 21st Century.* In: *Scientific American* 265 (1991), Nr. 3, S. 94–104

[Wei93] WEISER, Mark: *Some Computer Science Issues in Ubiquitous Computing.* In: *Communications of the ACM* 36 (1993), Nr. 7, S. 75–84

[WGC+11] WEAVER, James L.; GAO, Weiqi; CHIN, Stephen; IVERSON, Dean; VOS, Johan: *Pro JavaFX 2: A Definitive Guide to Rich Clients with Java Technology:* Apress, 2011. – ISBN 978-1430268727

[Wen12] WENGER, Rolf: *Handbuch der .NET 4.0-Programmierung: Verteilte Anwendungen :* Microsoft Press, 2012 (3)

[WTK+04] WALSH, William E.; TESAURO, Gerald; KEPHART, Jeffrey O.; DAS, Rajarshi: Utility Functions in Autonomic Systems. In: *Proceedings of the 1st International Conference on Autonomic Computing.* 17-19 May 2004, New York, NY, USA. Washington, DC, USA: IEEE Computer Society Press, 2004, S. 70–77

Abbildungsverzeichnis