Inhaltsverzeichnis

1 Whole Part Pattern

1.1 Intro

Das Whole-Part Pattern zeigt sich dann, wenn ein Objekt sich aus mehreren Zusammensetzt / mehrere Objekte beinhaltet. Es hilft dabei, Komponenten welche eine semantische Einheit bilden zu gruppieren und zu aggregieren, und orchestriert die Zusammenarbeit ebendieser Komponenten.

1.2 Problem

Eine Gruppierung / Ansammlung von Objekten hat möglicherweise emergente Eigenschaften, also Eigenschaften welche sich nur dann ergeben, wenn gewisse Objekte speziell zusammenarbeiten.

1.3 Forces

Ein komplexes Objekt sollte entweder in einzelne, kleinere Objekte zerlegt werden, oder aus bestehenden Objekten zusammengesetzt werden.

Diese komplexen Objekte sollen von aussen als eine einzige, atomare Einheit wahrgenommen werden, sie sollen also keinen direkten Zugriff auf die darunterliegenden Objekte ermöglichen.

1.4 Solution

Es wird eine Struktur eingeführt welche kleinere Objekte / Strukturen umfassen kann. Für diese Aggregation wird ein Interface definiert welches die einzige Möglichkeit ist auf die Funktionalitäten der enthaltenen Objekte zuzugreifen.

Es gibt dabei die folgenden 3 Arten von Relationships:

1.4.0.1 assembly-parts

Alle Parts sind fest integriert und in die interne Struktur eingebunden. Parts können nicht dynamisch entfernt oder hinzugefügt werden.

1.4.0.2 shared-parts

Parts können von mehreren Wholes gleichzeitig genutzt werden. Sie sind darauf angewiesen, dass ihr Life-cycle von einem zentralen komponenten gehandhabt wird, oder müssen ihren Life-cycle selbst managen.

1.4.0.3 container-contents

Die Parts sind weniger fest integriert als in *assembly-parts*, und das Whole erlaubt eine dynamischere Handhabung. So ist es vorgesehen, dass zur Laufzeit Teile des Inhalts ausgetauscht, hinzugefügt oder entfernt werden.

1.4.0.4 collection-members

Die Parts sind 'ähnliche' Objekte und werden durch das Whole gruppiert. Funktionalitäten wie Iterieren oder eine Aktion für alle / auf allen Parts durchzuführen. Es gibt keine Unterscheidung zwischen den enthaltenen Parts, es sind also alle Gleichwertig und haben keine speziellen Funktionen oder Eigenschaften

1.4.1 Structure

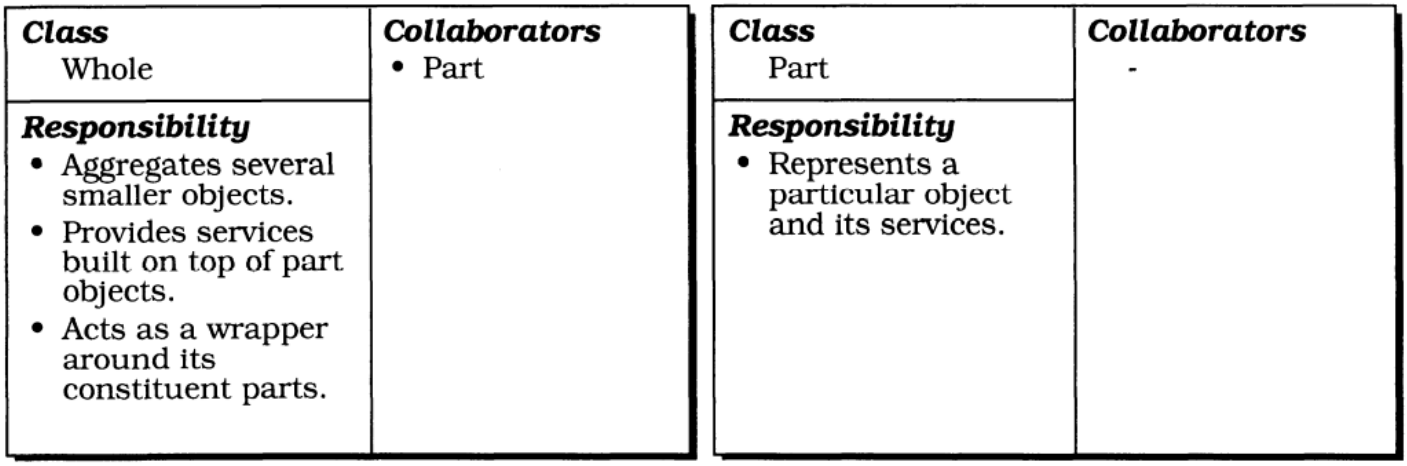

Abbildung 1: Whole-part Structure

1.5 Consequences

1.5.0.1 Vorteile

- Austauschbarkeit der Komponenten
  - das ganze Whole kann neu implementiert werden ohne den Client zu verändern
- Aufteilung der Verantwortlichkeiten
  - einzelne Verantwortlichkeiten sind an einzelne Parts delegiert
- Wiederverwendbarkeit
  - einzelne Parts / ganze Wholes können wiederverwendet werden

1.5.0.2 Nachteile

- Weniger effizient aufgrund Indirektion
  - Overhead zur Laufzeit (verglichen mit einem Monoliten)
- Komplexität der Zerlegung von Funktionalität in einzelne Teile

1.6 Known Uses

- OO-Klassenbibliotheken / Frameworks stellen collection-members Anwendungen zur Verfügung (lists, sets, maps, etc.)
- Eine Spezialisierung des Whole-Part Patterns ist das Composite Pattern nach GoF, welches erlaubt, dass Whole und Part identisch behandelt werden können. Dies setzt jedoch voraus, dass Whole und Part ein identisches Interface besitzen.

1.7 Wichtige Eigenschaften

- Interface von aussen atomar
  - kein Zugriff auf Komponenten welche darin enthalten sind
- ermöglicht emergente Eigenschaften / Verhaltensweisen

1.8 Examples

Das Whole-Part Pattern tritt extrem oft auf, z.B. in

- HttpClient (https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpClient.html)
- Als Spezialisierung (Composite Pattern) in GUI-Libraries
  - Java AWT
  - WPF
  - Android
- OO-Libraries (collection-member)
  - Lists
  - Sets
  - Maps

1.9 Prüfungsfragen

1. Beim Whole-Part Pattern ist es wichtig, dass das Whole weiterhin den direkten Zugriff auf die Parts ermöglicht. [Nein]
2. Lists, Sets und Maps sind eine Anwendung des *collection-member* Whole-Part Pattern. [Ja]

2 Forwarder-Receiver Pattern

2.1 Intro

Nur wenige Systeme laufen auf einem einzelnen Computer. Viele sind darauf angewiesen zu kommunizieren. Das Forwarder-Receiver Design Pattern ermöglicht eine abstrahierte Kommunikation über peer-to-peer Verbindungen (Jedes System kann als Client oder Server agieren).

2.2 Problem

Beim bauen von verteilten Systemen werden aufgrund von Performance oft low-level Komponenten verwendet anstatt höher abstrahierte Konzepte zur inter-Prozess Kommunikation. Diese low-level Komponenten (z.B. Sockets) sind oftmals Betriebssystem abhängig und wenn genutzt schwierig zu ersetzen.

2.3 Forces

Besonders nützlich ist das Forwarder-Receiver Pattern wenn folgende Bedingungen zutreffen auf das System das gebaut werden soll:

- Der Kommunikationsmechanismus soll austauschbar sein.
- Das Kommunikationsmodell ist peer-to-peer.
- Die Performanz der Kommunikation ist für die Applikation wichtig.

2.4 Solution

Alle Kommunikationsdetails werden in separate Komponenten ausgelagert. Dazu werden drei verschiedene Arten von Komponenten benötigt, **Forwarders**, **Receivers** und **Peers**. Die Kommunikation zwischen den Peers findet wie auf der Abbildung unten ersichtlich nur noch via Forwarder und Receiver statt.

2.4.1 Structure

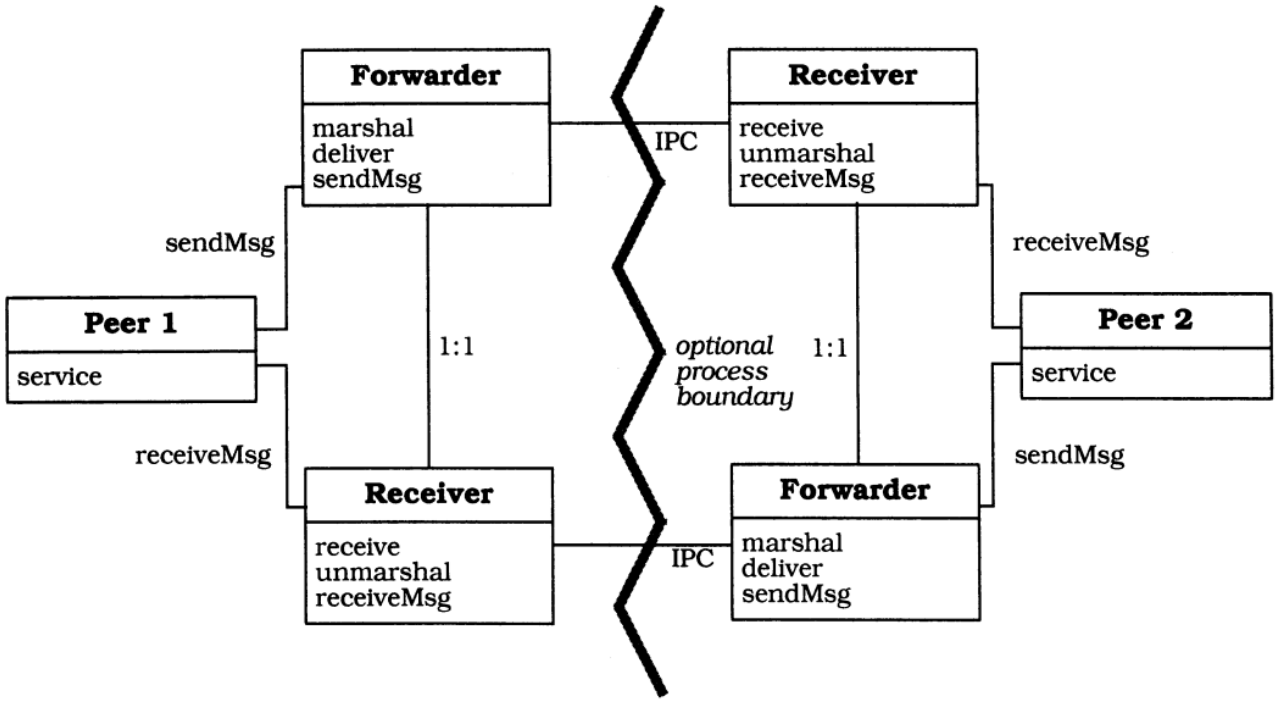

Abbildung 2: Forward-receiver Structure

2.4.1.1 Peers

Peers sind Teile einer Applikation die während ihrer Ausführung mit anderen Peers kommunizieren. Jeder Peer kennt den Namen oder die Adresse (Variante Forwarder-Receiver without name-to-address mapping) der Peers mit welchen er kommunizieren möchte. Der Peer verwendet Forwards um Nachrichten zu versenden und Receiver um Nachrichten zu empfangen. Peers sind Prozesse die sich auf derselben oder unterschiedlichen Maschinen befinden.

2.4.1.2 Forwarders

Forwarder werden zum Versenden von Nachrichten verwendet und bieten eine Abstraktion für einen IPC (interprocess communication) Mechanismus an. Die Details des IPC Mechanismus sind dem Aufrufer des Forwarder nicht bekannt. Forwarder werden via `sendMsg` aufgerufen, dazu muss der Aufruf den Namen des Receiver angeben.

Forwarder verfügen über folgende interne Funktionalität:

- Mapping von Namen zu physischen Adressen
- Funktionalität zum Umwandeln der Nachricht in etwas das vom IPC Mechanismus verstanden wird (`marshal`)
- Funktionalität zum Versenden der Nachricht (`deliver`)

2.4.1.3 Receivers

Receiver sind das Gegenstück vom Forwarder und empfangen Nachrichten. Receiver bieten ebenfalls eine Abstraktion über einen IPC Mechanismus, so dass die Aufrufer des Receiver nichts über die Details des IPC Mechanismus wissen müssen. Ein Aufrufer teilt dem Receiver via `receiveMsg` mit das eine Nachricht erwartet wird, sobald die Nachricht eingetroffen ist leitet der Receiver die Nachricht dem Aufrufer weiter.

Receiver verfügen über folgende interne Funktionalität:

- Funktionalität zum empfangen von Nachrichten (`receive`)
- Funktionalität zum umwandeln der empfangenen Nachrichten zu etwas das vom Aufrufer verstanden wird (`unmarshal`)

2.4.2 Ablauf von Messageübertragung

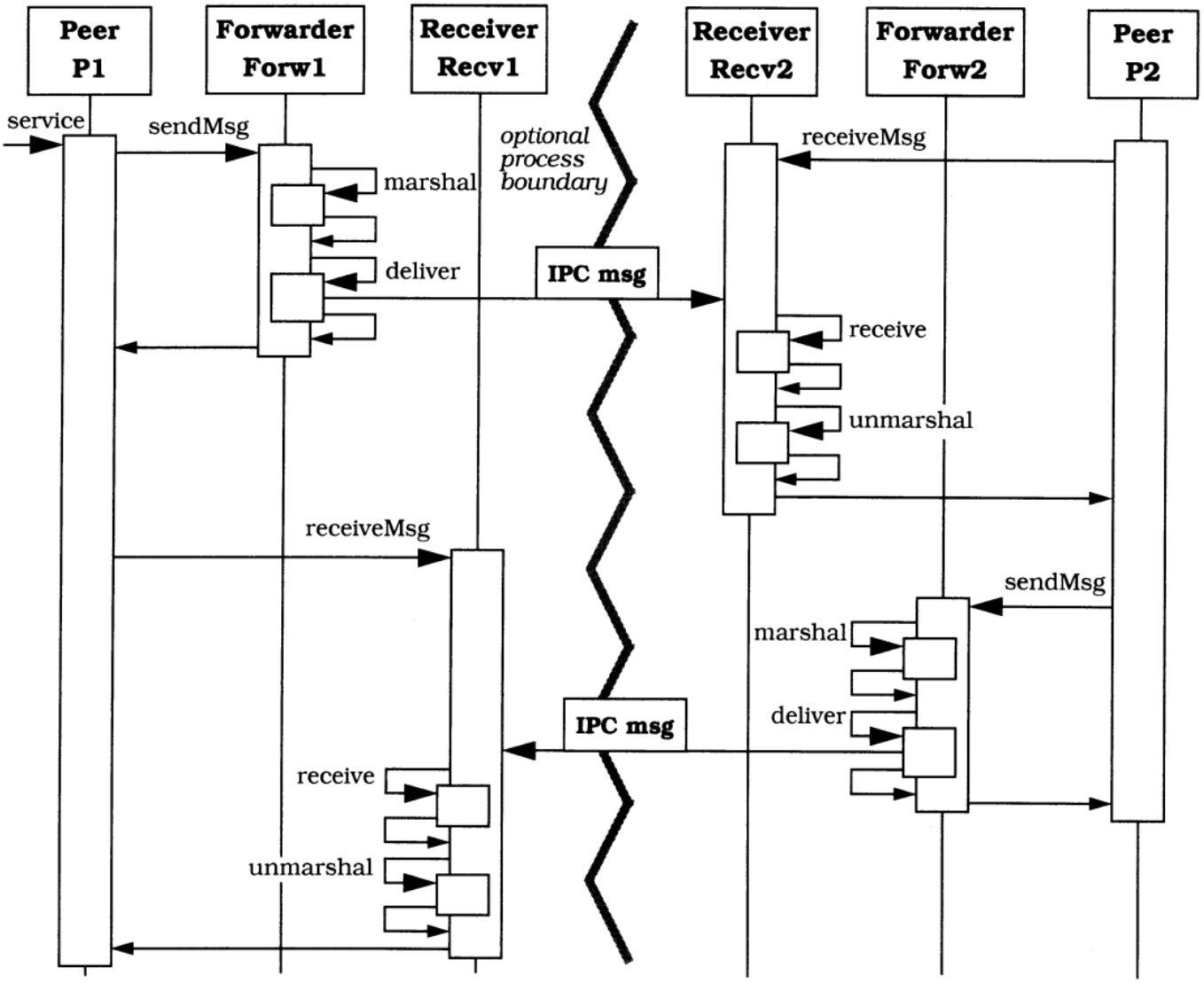

Abbildung 3: Forwarder-Receiver

2.5 Consequences

2.5.0.1 Vorteile

- Effiziente IPC
  - die Kommunikation zwischen Prozessen wird strukturiert
  - `Forwarders` kennen die physikalische Erreichbarkeit der `Receivers`, entsprechend müssen Remotecomputer nicht erst gefunden werden
- IPC relevante OS Abhängigkeiten sind in `Forwarders` und `Receivers` abgekapselt

2.5.0.2 Nachteile

- Flexible Rekonfiguration von Komponenten ist nicht möglich
- Falls die Verteilung von Peers sich zur Laufzeit ändern ist es schwer Anpassungen vorzunehmen
  - *(dies kann mittels Central-Dispatcher [siehe Client-Dispatcher-Server Pattern] gelöst)*

2.6 Wichtige Eigenschaften

- Peers müssen die konkreten Mechanismen zur Kommunikation nicht kennen

- IPC Implementation ist flexibel austauschbar

2.7 Examples

Konkrete Beispiele für das Forwarder-Receiver Pattern konnten wir nicht finden. Wir gehen aber stark davon aus, dass dieses Pattern - besonders auch weil es naheliegend und elegant ist - in vielen Softwarelösungen angewendet wird.

2.8 Prüfungsfragen

1. Das Forwarder-Receiver Pattern nutzt eine hohe Abstraktionsebene der Netzwerkkommunikation [Nein]
2. Das Forwarder-Receiver Pattern ist für eine peer-to-peer Kommunikation ausgelegt [Ja]

3 Blackboard Pattern

3.1 Intro

Das Blackboard Architekturpattern wird für Probleme verwendet für die es keine deterministische Lösungsstrategie gibt. In einem Blackboard arbeiten verschiedene Subsysteme zusammen und generieren eine Lösung. Die Lösung kann unvollständig oder eine Annäherung sein. Eine Analogie für das Pattern ist das gemeinsame Lösungsfinden unter verwendung eines Blackboards, jeder Teilnehmer (Subsystem) an diesem Lösungsfindungstreffen hat Zugriff auf das Blackboard. Jeder Teilnehmer kann seine Gedanken äussern und das Blackboard anpassen. Indem das Blackboard ständig von immer anderen Teilnehmern verändert wird erarbeitet man gemeinsam eine Lösung.

3.2 Problem

Das Blackboard Pattern kann für Probleme verwendet werden für die es keine machbaren deterministischen Lösung gibt um raw data in eine high-level Struktur umzuwandeln. Unter high-level Struktur versteht man z.B. Diagramme oder Tabellen .

Typische Beispiele sind:

- Vision
- Image recognition
- Speech recognition
- Surveillance

Probleme für welche das Blackboard Pattern angewendet werden kann, lassen sich meistens in Teilprobleme zerlegen. Die Teilprobleme hängen oft nicht zusammen und können aus verschiedenen Fachgebieten stammen. Die Teilproblemlöser erwarten oft unterschiedliche Repräsenetation der Daten oder unterschiedliche Muster in den Daten.

Der grosse Unterschied zu einer funktionalen Vorgehensweise ist, dass die Teilproblemelöser nicht nach einer vorgegebenen Reihenfolge durchgeführt werden können.

Jede Durchführung eines Teilproblemlösers generiert eine Lösung sowie optional auch alternative Lösungen, das System muss sich dann darum kümmern, dass eine optimale Lösung ausgewählt wird.

Wichtig:

- Immer die Einschränkungen des Systems dokumentieren
- Wichtige Entscheidungen verifizieren

Artiffial Intelligence (AI) Systeme werden oft zum Lösen solcher Probleme verwendet, es gibt aber drei Gründe warum das zu mangelhaften Lösungen führt:

- Alle Teilprobleme müssen mit derselben Datenrepräsentation gelöst werden, obwohl es sich um Teilprobleme unterschiedlicher Fachgebiete handelt, die mit unterschiedlichen Datenrepräsentation aufgerufen werden sollten.
- Lösungen werden nur von einer Komponente (inference engine) getroffen, obwohl Teilproblemem mit unterschiedlichen Repräsentationen eigene Komponenten zur Lösungsfindung benötigen.
- AI Systeme folgen aus einem Tree, eine Lösungsfindung / Suche findet immer in diesem Tree statt und anhand von Attributen / Wissen wird ein vorgegebener Weg zu einem Leaf gefunden. Für ein Blackboard ist es wichtiger das Wissen dynamisch verändert werden kann und sich dadurch die Lösung verändert, dass kann mit einer AI nicht abgebildet werden.

3.3 Forces welche die Lösungen beeinflussen

- Das finden aller Lösungen kann nicht in absehbarer Zeit durchgeführt werden.
- Für ein Teilproblem können unterschiedliche Algorithmen angewendet werden.
- Unterschiedliche Algorithmen die sich überschneiden.
- unterschiedliche Repräsentationen von Input und Resultat, sowie Algorithmen die nach unterschiedlichen Mustern implementiert sind.
- Abhängigkeit zwischen Algorithmen, ein Algorithmus arbeitet mit dem Resultat eines anderen weiter.
- Angenäherte Daten werden verwendet.
- Algorithmen werden parallel durchgeführt.

3.4 Solution

Die Idee des *Blackboard Patterns* ist eine Ansammlung von unabhängigen Programmen welche gemeinsam koorperativ eine geteilte Datenbasis bearbeiten. Jedes dieser Programme ist auf einen bestimmten Teil der Lösungsfindung spezialisiert. Eine zentrale Kontrollkomponente orchestriert die Zusammenarbeit und den Ablauf der Programme.

Die Lösungsvorschläge (*Hypothesis*) der einzelnen Programme werden im Verlauf der Lösungssuche vereint, verändert oder sogar zurückgewiesen.

3.4.1 Struktur

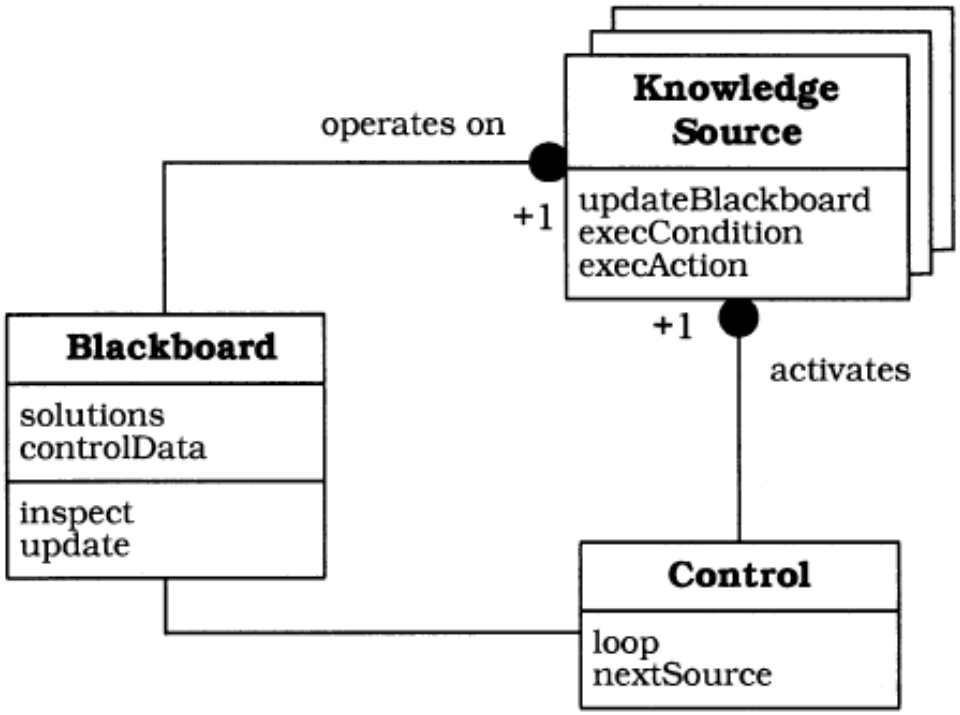

Abbildung 4: Blackboard Struktur

3.4.2 Blackboard

Das Blackboard ist der zentrale Datenspeicher und umfasst Elemente aus dem Lösungsraum und Kontrolldaten, das **Vokabular**. Es stellt eine Schnittstelle zur Verfügung mit welcher die **Knowledge Source**s Daten Lesen und Schreiben können, vor allem Lösungsvorschläge. Ein Lösungsvorschlag welcher zur Laufzeit von einer **Knowledge Source** geschrieben wird, wird als **Hypothesis** bezeichnet. Später verworfene Lösungsvorschläge werden vom Blackboard entfernt.

3.4.3 Hypothesis

Lösungsvorschläge haben eine 'Abstraktionsstufe', wobei höhere Stufen näher an der resultierenden Lösung sind, und tiefere Stufen näher am ursprünglichen Input in das System. Ebenfalls weisen sie eine *Credibility* aus, also wie wahrscheinlich es ist, dass der Lösungsvorschlag korrekt ist.

3.4.4 Knowledge Sources

Diese Komponenten sind getrennte, unabhängige Subsysteme welche auf einzelne Aufgaben spezialisiert sind. Sie können die Lösung nicht alleine finden, aber die Kombination der Vorschläge verschiedener Komponenten führt schlussendlich zur Lösungsfindung. Knowledge Sources kommunizieren nicht direkt miteinander, Datenaustausch findet nur mit dem Blackboard statt. Jede Source ist dafür verantwortlich anzugeben ob sie mit dem aktuellen Datenstand zur Lösung beitragen kann oder nicht.

3.4.5 Control

Die Control überwacht Änderungen auf dem Blackboard und entscheidet welche **Knowledge Source** als nächste zum Zug kommt. Diese Entscheidung wird aufgrund des Datenstandes auf dem Blackboard getroffen. Es ist möglich, dass die Control auf einen Zustand trifft in dem keine der Knowledge Sources angewendet werden kann - dann scheitert das System und kann kein Resultat liefern. Meist ist es jedoch eher so, dass (da einzelne Anwendungen von **Knowledge Source**s zu mehreren Lösungsvorschlägen führen) eher das Gegenteil das Problem ist - also dass zu viele Optionen vorhanden sind und die Ausführung ebendieser eingeschränkt werden muss.

Eine spezialisierte **Knowledge Source** entscheidet, wann das System fertig ist und welches das schlussendliche Resultat ist.

3.5 Implementation

Die folgenden Schritte zur Implementation empfehlen sich:

1. Definieren des Problems

   Wie sieht der Input aus? Sind Muster erkennbar? Gibt es Rauschen in Input? Welcher Output soll erreicht werden? Wie kann sichergestellt werden, dass die Resultate vertrauenswürdig sind?

2. Definieren des Lösungsraums

   Wie kann eine 'top-level' Lösung erkannt werden? Welche Abstraktionsstufen im Lösungsverlauf kann es geben? Kann eine Lösung aufgeteilt werden und unabhängig weiterverarbeitet werden?

3. Aufteilung der Lösungfindung in Schritte

   Wie werden Lösungen zu 'higher-level' Lösungen verfeinert? Welches Wissen wird benötigt um Teile des Lösungsraums auszuschliessen?

4. Aufteilung des Wissens in spezializierte **Knowledge Source**s

5. Definieren des **Vokabulars**

6. Spezifizieren der Kontrollkomponente

   Welchen Lösungsvorschlägen kann vertraut werden?

7. Implementieren der **Knowledge Source**s

3.6 Consequences

Das Blackboard pattern kann alle Forces adressieren und diese elegant lösen.

3.6.1 Vorteile

- **Experimentieren:** Probleme ohne konkreten Lösungsansatz können ohne alle möglichkeiten auszuprobieren gelöst werden.
- **Maintainability / Austauschbarkeit:** Durch die modulare struktur und unabhängigkeit der Komponenten können einzelne Teile einfach erweitert und ausgetauscht werden.
- **Wiederverwendbarkeit:** Knowledge Sources können für andere Probleme wiederverwendet werden.
- **Robustheit / Fehlertoleranz:** Jegliche Resultate sind hypothesen und verschiedene Pfade werden gegeneinander gewichtet um die optimale Lösung zu finden.

3.6.2 Nachteile

- **Testing:** Die Resultate sind nicht reproduzierbar und folgen keinem deterministischen algorithmus.
- **Lösung nicht garantiert:** Nicht alle aufgaben können gelöst werden.
- **Gute Control Strategy schwierig zu finden:** Ein experimenteller ansatz wird jenachdem benötigt um eine Control Strategy zu finden.

- **Effizienz**: Durch zurückgewiesene hypothesen die ausprobiert wurden ist die effizienz eines Blackboard systems tiefer als bei einem deterministischen algorithmus
- **Hoher aufwand**: Die entwicklung eines Blackboard systems ist sehr aufwändig.
- **Paralellisierbarkeit:** Paralellisierung vom pattern nicht angedacht, aber theoretisch möglich.

3.7 Wichtige Eigenschaften

- Undeterministischer Ansatz (Heuristisch)
- Für Komplexe Aufgaben ohne definierten Lösungsweg.
- Löst Teilprobleme und kann diese Lösungsansätze aufeinander aufbauen.
- Kein einheitliches Format der Daten benötigt für die verschiedenen Knowledge Sources.

3.8 Examples

- **HEARSAY II**: Dies ist eher ein historisches Projekt und ist der Ursprung dieses Patterns. Es wurde genutzt für eine der ersten Sprach-Verständnis Systeme und wurde 1980 publiziert.
- **CRYSALIS**: Eine Software benutzt um die dreidimensionale struktur von Protein Molekülen aufgrund von Röntgenstrahlen beugung herauszufinden.

3.9 Prüfungsfragen

1. Knowledge Sources interagieren direkt miteinander um gemeinsam eine Lösung zu finden. [Nein]
2. Das Blackboard Pattern nutzt einen Deterministischen Ansatz. [Nein]

4 Lazy Acquisition Pattern

4.1 Intro

Das **Lazy Acquisition Pattern** verschiebt das Laden von externen Ressourcen auf den spät möglichsten Zeitpunkt. Dies führt zu einer optimierten Verwendung von Ressourcen.

4.2 Problem

Ein System mit limitierter Verfügbarkeit von Hardwareressourcen oder Zeit muss hohen Ansprüchen gerecht werden. So sollen externe Inhalte schnell und zeitgerecht Verfügbar sein.

Moderne Software beinhaltet oft eine Vielzahl an externen Ressourcen in Form von Bildern, Dokumenten oder grossen Sammlungen von Datensätzen. Wenn all diese Ressourcen zu Beginn der Ausführung geladen werden, so kann es sein, dass ein Programm sehr lange braucht bis es bereit zur Verwendung ist.

4.3 Forces

4.3.0.0.1 Verfügbarkeit

Die Ressourcenbeschaffung soll eingerichtet werden, dass Ressourcenengpässe minimiert werden, und die richtigen Ressourcen zur richtigen Zeit verfügbar sind.

4.3.0.0.2 Stabilität

Ressourcenengpässe in einem System können zu Instabilität führen, entsprechend sollten Ressourcen in einer Art beschafft werden, dass sie möglichst kleine Auswirkungen auf die Stabilität des Systems haben

4.3.0.0.3 Schneller Systemstart

Die Ressourcenbeschaffung zur Startzeit soll so gestaltet werden, dass die Startzeit minimeirt wird.

4.3.0.0.4 Transparenz

Die Lösung für diese Herausforderungen soll für Anwender transparent sein, also keinen Unterschied zur Ausgangslage machen.

4.4 Solution

Die Resources werden alloziert am letzten möglichen Zeitpunkt. Erst wenn auf die tatsächlichen Daten einer Resource zugegriffen wird, wird die Resource geladen. Vorher existiert ein Proxy object das beim eigentlichen Resourcen aufruf erstellt wird und die möglichkeit beinhaltet die tatsächliche Resource nachzuladen. Sobald die Resource geladen wurde, werden alle Anfragen an den Proxy direkt an die Resource weitergeleitet. Somit ist der Proxy komplett transparent gegenüber dem Endbenutzer.

4.5 Implementation

4.5.1 Model

Class **Resource User**	*Collaborator*
Responsibility • Acquires and uses a resource.	• Resource Proxy

Class **Resource**	*Collaborator*
Responsibility • Is acquired and used through the resource proxy.	

Class **Resource Proxy**	*Collaborator*
Responsibility • Pretends to be the resource. • Provides the same interface as the resource. • Makes the actual resource available from the resource provider.	• Resource • Resource Provider

Class **Resource Provider**	*Collaborator*
Responsibility • Manages and provides resources to resource proxies.	• Resource

Abbildung 5: Lazy Acquisition Model

- Ein “Resource User” alloziert und benutzt Resourcen.
- Ein “Resource Proxy” ist der Platzhalter für eine Resource und lädt diese sobald sie tatsächlich benötigt wird.
- Der “Resource Provider” managed verschiedene Resourcen und stellt diese zu Verfügung.

4.5.2 Szenarien

4.5.2.1 Resource Proxy erstellung

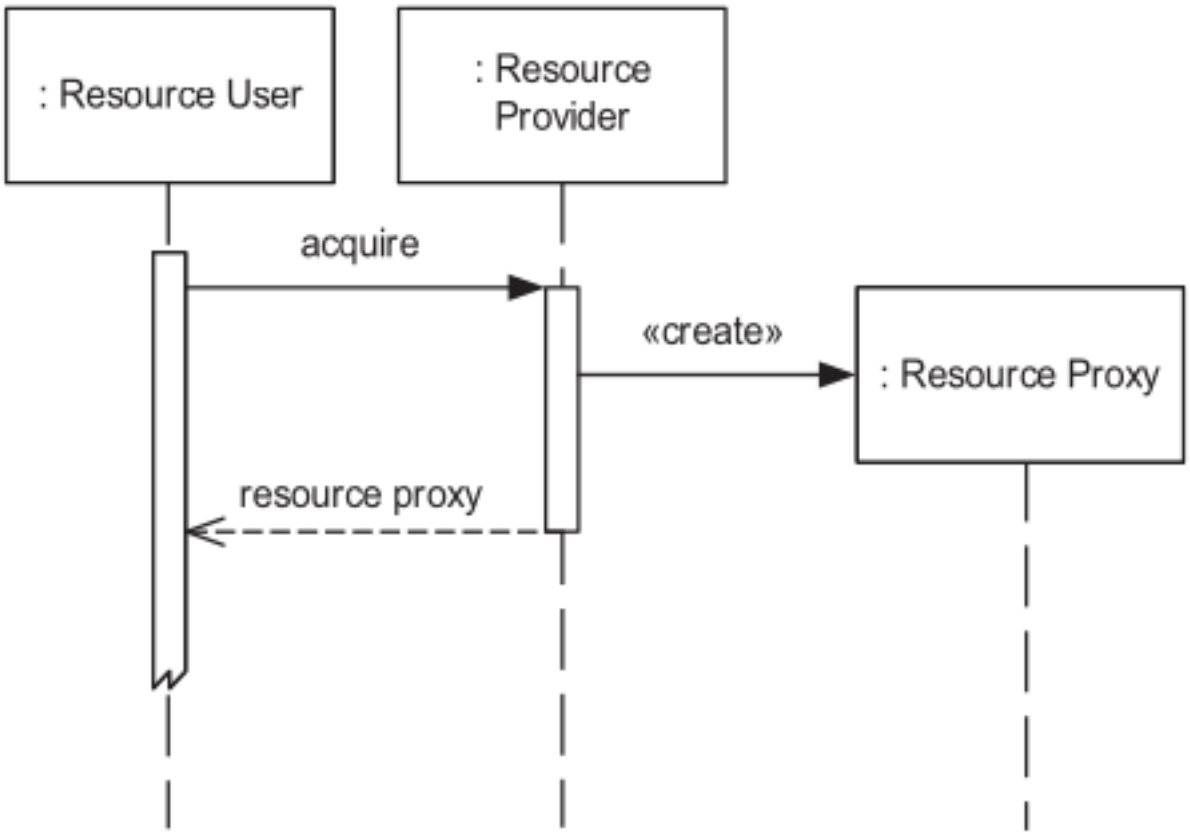

Abbildung 6: Lazy Acquisition Erstellung

Der “Resourcen Provider” ist für die Interaktion mit dem “Resource User” verantwortlich. Er agiert als Factory und erstellt die angefragen “Resource Proxies”.

4.5.2.2 Resource Proxy benutzung

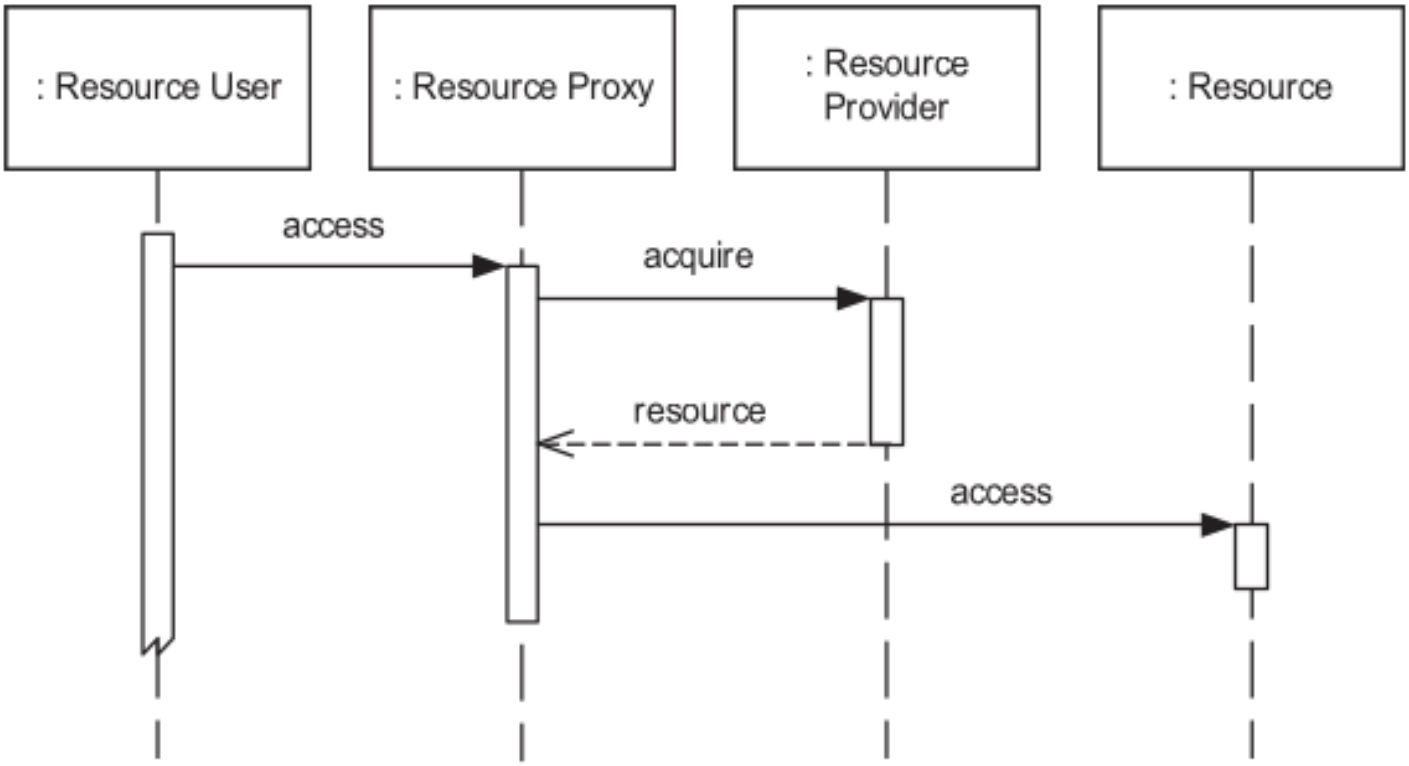

Abbildung 7: Lazy Acquisition Benutzung

Der “Resource User” kann nun über den Proxy auf die Resource zugreifen. Die allokation der tatsächlichen Resource wird wieder über den “Resource Provider” gemacht.

4.5.3 Steps

1. Identifizieren der Resourcen die “Lazy” geladen werden müssen.

- Resourcen die aufwändig sind zu allozieren
- Resourcen die eine limitierte auflage haben
- Resourcen die lange nach der allokation unbenutz bleiben

2. Proxy Interface definieren
3. Resource Proxy implementieren
4. Allokations strategie definieren
5. Resourcenfreigabe strategie definieren

4.6 Consequences

4.6.1 Vorteile

- Verfügbarkeit: Es werden nicht alle Ressourcen von Beginn angefordert, dadurch wird die Möglichkeit verringert, dass Ressourcen geladen werden die nicht gebraucht werden.
- Stabilität: Es wird sichergestellt das Ressourcen nur geladen werden, wenn sie benötigt werden.
- Optimaler Systemstart: Ressourcen die nicht sofort benötigt werden, werden erst zu einem späteren Zeitpunkt geladen, dadurch wird die Start zeit optimiert.
- Transparenz: Lazy Acquisiotion ist transparent gegenüber dem Benutzer der Ressource.

4.6.2 Nachteile

- Space overhead: Es wird wegen der Indirektion des Proxys zusätzlicher Speicher benötigt.
- Time overhead: Es kann zu einer Verzögerung kommen beim Anfordern von Ressourcen und die Indirektion verursacht ebenfalls eine Verzögerung beim Ausführen des Programms.

- Vorhersagbarkeit: Wenn mehrere Teile des Systems den Zugriff auf eine Ressource hinausschieben und alle zur selben Zeit den Zugriff machen ist die Zugriffszeit nicht vorhersagbar.

4.7 Wichtige Eigenschaften

- Zugriff auf eine Ressource wird nur gemacht wenn sie benötigt wird.
- Performance und Stabilität eines Systems können besser gewährleistet werden.
- Vorhersagbarkeit der Zugriffszeit kann nicht mehr gewährleistet werden.

4.8 Examples

- **Singleton**: Wird beim ersten Gebrauch initialisiert
- **Haskell**: Expressions werden lazy evaluiert.
- Weitere: J2EE, Ad hoc networking, Operating systems, .NET Remoting, COM+, JIT compilation, Java, Manufactoring, ...

4.9 Prüfungsfragen

1. Wenn alle Ressourcen zu Beginn geladen werden damit die optimale Performance gewährleistet werden kann, redet man von Lazy Acquisition? [Nein]
2. Lazy Acquisition führt dazu, dass die Startdauer eines Systems sehr viel grösser wird. [Nein]

5 Coordinator Pattern

5.1 Intro

Das Ziel des Coordinator Pattern ist es Konsistenz beizubehalten in einem System welches einen Task durch die Zusammenarbeit von mehreren Teilnehmern löst. Entweder wird der gesamte Task abgeschlossen also alles Teilnehmer lösen ihr Problem erfolgreich oder im Falle eines Fehlers bei einem Teilnehmer wird alles auf einen konsistenten Zustand zurückgesetzt, der Zustand bevor der Task begonnen wurde.

Ein typisches Beispiel sind Datenbank Transaktionen.

5.2 Problem

Oft verwenden Systeme zum ausführen eines Tasks mehrere Teilnehmer. Ein Teilnehmer kann in diesem Fall ein Ressource user und ein Ressource provider sein oder auch eine aktive Ressource wie ein Service, welcher am Task beteiligt ist. Teilnehmer laufen auf:

- demselben Prozess wie das System,
- mehreren erzeugten neuen Prozessen,
- sowie auf mehreren Nodes.

Jeder Teilnehmer ist für die Durchführung eines Teil des Tasks zuständig, die Teilnehmer werden nach einer fixen Reihenfolge aufgerufen.

Damit der Task erfolgreich abgeschlossen werden kann müssen alle Teilnehmer ihren Teiltask erfolgreich abschliessen. Nach dem erfolgreichen abschliessen eines Tasks ist das System in einem **konsistenten** Zustand. Falls aber ein Fehlerauftritt und ein Teilnehmer nicht erfolgreich abgeschlossen hat ist das System in einem **inkonsistenten** Zustand, da bereits alle erfolgreich Abgeschlossenen Teilnehmer von einem neuen Zustand ausgehen aber alle fehlgeschlagenen oder noch nicht ausgeführten Teilnehmer vom alten Zustand ausgehen.

Ein inkonsistenter Zustand führt dazu, dass das Ausführen von weiteren Tasks fehlschlägt oder zu einen falschen Ergebnis führt.

5.3 Forces

Damit man dieses Problem lösen kann muss folgendes beachtet werden.

- Konsistenz: Ein Task erstellt entweder einen neuen validen Zustand oder setzt alles zurück auf Zustand bevor der Task ausgeführt wurde.
- Atomarität (Abgeschlossenheit): In einem Task der 2 oder mehr Teilnehmer hat müssen alle Teilnehmer abgeschlossen sein oder keiner, auch wenn sie nicht voneinander abhängig sind.

- Location transparency: Die Lösung muss auch in verteilten Systemen funktionieren.
- Skalierbarkeit: Die Lösung muss skalieren mit der Anzahl Teilnehmer, sod das sich die Performance nicht signifikant verschlechtert.
- Transparenz: Die Lösung muss transparent sein dem Benutzer gegenüber und sollte ein Minimum an Codeänderungen erfordern.

5.4 Solution

Die Lösung ist das Einführen eines **Coordinator**s. Der Coordinator ist verantwortlich für Ausführung und Fertigstellung eines *Tasks* durch alle Teilnehmer.

Um dies zu erreichen, wird der Ablauf dabei in 2 Phasen aufgeteilt; die `prepare` Phase, und die `commit` Phase.

5.4.0.1 Prepare Phase

In dieser Phase weist der Coordinator alle Teilnehmer an ihre Arbeiten vorzubereiten. Dabei ist jeder Teilnehmer angehalten die Konsistenz zu überprüfen, und sicherzustellen, dass das Ausführen einer Aufgabe nicht scheitern würde.

Falls nach dieser Phase ein (oder mehrere) Teilnehmer **nicht** erfolgreich sind, bricht der Coordinator die Ausführung ab und weist alle erfolgreichen Teilnehmer an die Ausführung ihrer (Teil-)Aufgabe abzubrechen und den Zustand vor Beginn der Aufgabe herzustellen.

5.4.0.2 Commit Phase

Diese Phase wird nur erreicht, wenn alle Teilnehmer die Prepare Phase erfolgreich abgeschlossen haben.

Der Coordinator kann sich jetzt also sicher sein, dass alle Teilnehmer die Aufgabe erfolgreich bearbeiten können, und weist alle Teilnehmer an ebendies zu tun.

5.4.1 Struktur

Die folgende Aufstellung gibt einen Überblick:

Class **Task**	*Collaborator*
Responsibility • Unit of work	

Class **Participant**	*Collaborator*
Responsibility • Does part of the work of a task • Registers with the Coordinator • Includes resource users, resource providers, and resources	• Task • Coordinator

Class **Coordinator**	*Collaborator*
Responsibility • Coordinates the consistent completion of a task	• Participant

Class **Client**	*Collaborator*
Responsibility • Initiates a task • Commits a task	• Coordinator • Task

Abbildung 8: Coordinator Struktur

Grundsätzlich gibt es 2 Interaktionspunkte zwischen den möglichen Komponenten.

1. Der Client weist den Coordinator an, eine Aufgabe durchzuführen
2. Der Coordinator kommuniziert mit dem Teilnehmern

5.4.1.1 Erfolgreicher Ablauf:

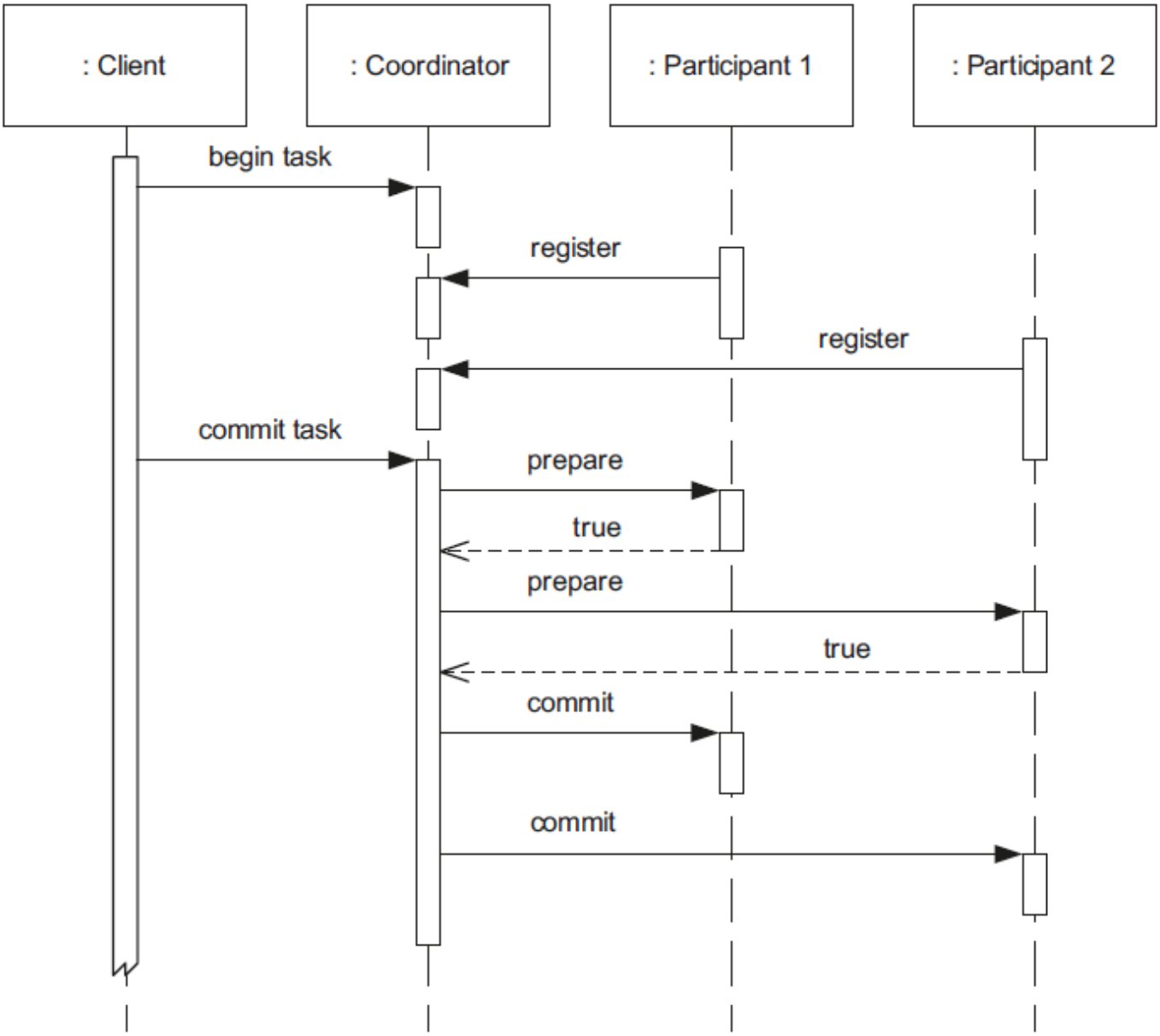

Abbildung 9: Coordinator Ablauf

5.4.1.2 Ablauf mit Abbruch:

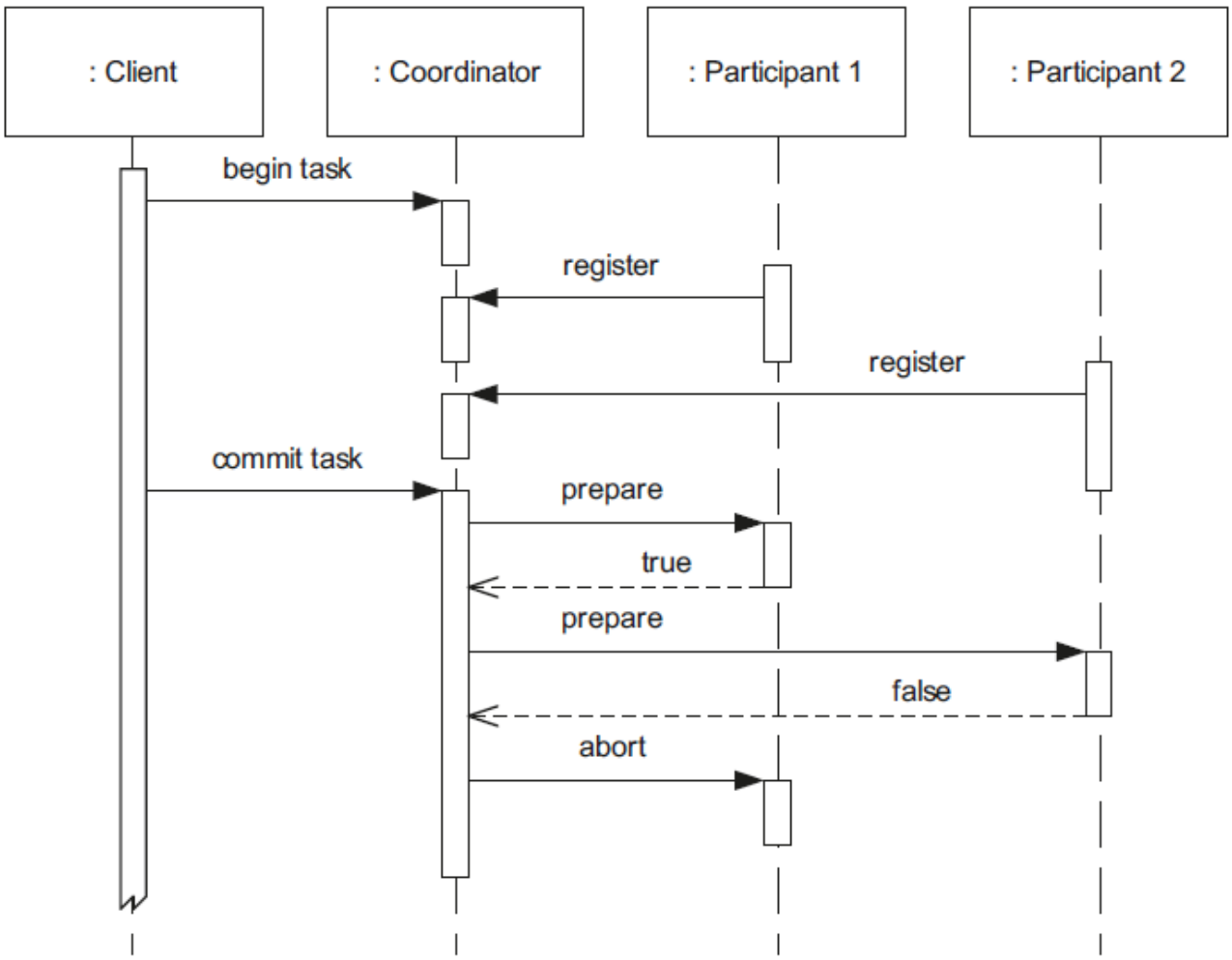

Abbildung 10: Coordinator Ablauf mit Abbruch

5.5 Implementation

Um das Coordinator Pattern grundlegend zu implementieren sind 3 Schritte nötig.

1. *Teilnehmer identifizieren* Teilnehmer deren Aufgaben koordiniert werden sollen, müssen identifiziert werden. Teilnehmer können dabei Ressourcen, Ressourcenanbieter oder Ressourcennutzer sein.

2. *Koordinationsschnittstelle definieren* Dieses Interface wird von allen Teilnehmern welche Arbeiten ausführen implementiert sein. Es stellt die Grundlage für die Fähigkeiten des Coordinators dar. Dieses interface könnte wie folgt aussehen:

```
public interface Coordination {
 boolean prepare ();
 void abort();
 void commit();
}
```

3. *Koordinatorschnittstelle definieren* Dieses Interface ermöglicht Clients eine Aufgabe zu beginnen oder abzuschliessen. Dieses interface könnte wir folgt aussehen:

```
public interface Coordinator {
    void beginTask();
    void register(Coordination participant);
    boolean commitTask();
}
```

Wenn der Client `beginTask()` aufruft passiert vorerst noch nichts - die `participant`s müssen sich erst noch registrieren. Sobald alle `participant`s sich registriert haben, ruft der Client `commitTask()` auf. Dies zeigt dem Coordinator, dass alle Teilnehmer vorhanden sind, und dass er damit beginnen kann die Aufgabe mit den Teilnehmern auszuführen.

5.5.1 Varianten

Third-Party Registration

Die Registrierung auf dem Coordinator kann bei dieser Variante durch eine Drittpartei geschehen, und nicht zwingend durch die Teilnehmer. Dies kann nützlich sein, wenn der Client ggf. kontrollieren können muss weile Teilnehmer sich an einer Aufgabe beteiligen.

Participant Adapter

Diese Variante sieht vor, dass Teilnehmer nicht zwingend direkt das `Coordination` Interface implementieren müssen, stattdessen werden Teilnehmer mit einem Adapter versehen welche diese Aufgaben übernimmt. Diese Variante ist ggf. nützlich um *legacy* Code besser integrieren zu können.

Three-phase Commit

Diese Variante ermöglicht die Behandlung von Fehlern welche in der Commit-Phase auftreten. Ist diese Variante gewünscht, wird bei der Implementation ein 4ter Schritt nötig:

4. *Behandeln von Fehlern* Obwohl alle Teilnehmer in der `prepare` Phase eine mögliche Ausführung angezeigt haben, könnte es trotzdem vorkommen, dass (z.B. aufgrund Netzwerkproblemen) eine Aufgabe nicht abgeschlossen werden kann. In diesem Error-Handling könnte der Coordinator ggf. eine `rollback()` Methode auf den erfolgreichen Teilnehmern ausführen um diesen die Möglichkeit zu geben gemachte Änderungen rückgängig zu machen.

5.6 Consequences

5.6.1 Vorteile

- **Atomarität**: Es wird sichergestellt, dass alle Aufgaben erfolgreich abgeschlossen werden oder sonst nichts gemacht wird. “All or nothing” prinzip.
- **Konsistenz**: Die konsistenz des Systems wird gewährleistet, da ein System von einem validen Zustand in den nächsten wechselt. Und falls das nicht möglich ist, wird das System im alten konsistenten Zustand belassen.
- **Skalierbarkeit**: Es spielt keine Rolle wie viele “Participants” existieren. Denn der Client interagiert nur über den Coordinator mit ihnen.
- **Transparenz**: Der Client kennt nur die Tasks und hat keine Kenntnis der 2-Phasen. Die Konsistenz und Atomarität ist transparent umgesetzt gegenüber dem Client.

5.6.2 Nachteile

- **Overhead**: Jeder Task wird in zwei Phasen unterteilt. Die überprüfung in der ersten Phase und die ausführung danach stellen einen Overhead dar.
- **Zusätzliche Verantwortung**: Die registrierung der “Participants” beim “Coordinator” stellt eine zusätzliche Verantwortung dar welche die Participants erfüllen müssen.

5.7 Wichtige Eigenschaften

- Das Pattern umfasst 2-Phasen welche nacheinander aufgerufen werden. Nur wenn eine problemlose ausführung möglich ist, wird diese auch umgesetzt.
- Das System befindet sich immer in einem Validen zustand.
- Der Client hat muss sich nicht um die Konsistenz und Atomarität kümmern.

5.8 Examples

- **Java Authentication and Authorization Service (JAAS)**: Das Login in JAAS erlaubt dynamisch konfigurierbare login module welche alle abgeschlossen werden müssen für eine erfolgreiche Authentifizierung. Dies wurde mit 2-Phasen und einem Coordinator gelöst.
- **2-Phase Commit Protokoll**: Datenbanken brauchen das Coordinator pattern und das 2-Phase Commit Protokoll für die synchronisation von mehreren Datenbanksystemen.

- **Software Installationen**: viele Installationsprogramme brauchen eine Form des Coordinator Patterns um sicherzustellen das eine installation erfolgreich sein wird. Z.B. das sie erst überprüfen ob genug speicherplatz verfügbar ist.

5.9 Prüfungsfragen

- Der Client gibt Tasks direkt an einen Participant welcher diesen dann für ihn atomar ausführt. [Ja | Nein]
- Das Ziel des Coordinator Patterns ist "All or nothing" [Ja | Nein]

6 Resource Lifecycle Manager

6.1 Intro

Die Frage wer nun schlussendlich verantwortlich ist und wann die Instanziierung und der Abbau von Ressourcen in einer komplexen Applikation geschieht kann verschieden beantwortet werden. Es könnte die Ressource selbst sein die sich abbaut oder auch der Benutzer der Ressource.

Man kann aber auch eine Komponente entwickeln welche speziell diese Responsibility übernimmt. Diese Komponente ist der Resource Lifecycle Manager.

6.2 Problem

Ressourcen können von verschiedensten Typen sein, zum Beispiel Netzwerkverbindungen, Threads, Workers, oder auch Synchronisationsobjekte wie Mutexe und Semaphore. Wenn das Management all dieser Ressourcen in einem grossen System jeweils in den Aufrufern oder auf den Ressourcen geschieht geht der Überblick sehr schnell verloren.

Beim Beispiel der Netzwerkverbindungen geschehen diese meist synchron, heisst ein Client schickt an einen Server und dieser sendet eine Antwort. Wenn diese Kommunikation aber nun Stateful ist, und je nachdem sogar asynchron nach einer gewissen Zeit vom Server eine Antwort an einen Client geschehen muss wird das Handling dieser Verbindungen immer schwieriger. Da solche Applikationen oft mit tausenden Verbindungen zu tun haben, ist es wichtig das solche, die nicht mehr relevant sind, auch wieder zugemacht werden, denn sonst leidet die Stabilität des Systems. Irgendwo muss nun dieses Management all dieser Verbindungen gemacht werden.

6.3 Forces

Verfügbarkeit: Typischerweise wächst die Anzahl Ressourcen nicht gleichmässig mit der Grösse des Systems. Deshalb müssen Ressourcen in grossen Systemen effizient und effektiv verwaltet werden, damit die Verfügbarkeit gewährleistet werden kann.

Skalierbarkeit: Mit der Grösse des Systems steigt auch die Anzahl Ressourcen, dass kann das verwalten der Ressourcen erschweren.

Komplexität: Abhängigkeiten müssen genau beobachtet werden, damit Ressourcen rechtzeitig freigegeben werden können.

Leistung: Optimierungen sind wichtig damit alle "performance bottlenecks" eliminiert werden können, sind aber komplex wenn sie von einzelnen Ressource Usern angewendet werden.

Persistenz: Es darf nicht vergessen werden Ressourcen freizugeben und das die Verwendung der Ressourcen zu kontrollieren.

Abhängigkeiten: Ressourcen haben Abhängigkeiten auf andere Ressourcen und dadurch haben diese Abhängigkeiten auch einen Einfluss auf den Lebenszyklus der Ressource diese Abhängigkeiten müssen entsprechen kontrolliert werden.

Anpassungsfähigkeit: Ressourcen sollten flexibel verwaltet werden, so das verschiedene Strategien für das Verwaltungsverhalten gewählt werden können.

Transparenz: Das verwalten des Lebenszyklus sollte dem Ressourcenverwender gegenüber transparent sein.

6.4 Solution

Es wird ein Resource Lifecylce Manager (RLM) eingeführt der dafür zuständig ist die Ressourcen eines Ressourcenverwenders zu verwalten. Der RLM wird verwendet, um Zugriff auf spezifische Ressourcen zu erhalten und kann die Ressource, falls sie noch nicht vorhanden ist erstellen. Ebenfalls verfügt der RLM über Wissen über die Ressource und kann deshalb den Zugriff auch verweigern, z.B. wenn zu wenig Speicher vorhanden ist.

Ein RLM kann für einen oder mehrere Typen von Ressourcen zuständig sein und kann falls vorhanden mit Anhängigkeiten zu anderen Ressourcen umgehen. Die Abhängigkeiten werden entweder von einem zentralen RLM verwaltet, welcher die

Ressource und deren Abhängigkeiten verwaltet oder von einem separaten RLM der nur für das Verwalten der Abhängigkeiten zuständig ist.

Für die Lösung braucht es die folgenden Teilnehmer:

Class **Resource User**	Collaborator
Responsibility • Acquires and uses resources. • Releases unused resources to the resource lifecycle manager.	• Resource • Resource Lifecycle Manager

Class **Resource**	Collaborator
Responsibility • Represents a reusable entity, such as memory or a thread. • Is acquired from the resource provider by the resource lifecycle manager.	

Class **Resource Lifecycle Manager**	Collaborator
Responsibility • Coordinates lifecycle of resources including creation/acquisition, reuse, and destruction.	• Resource • Resource Provider

Class **Resource Provider**	Collaborator
Responsibility • Owns and manages several resources.	• Resource

Abbildung 11: Resource Lifecycle Teilnehmer

Die Teilnehmer interagieren wie folgt:

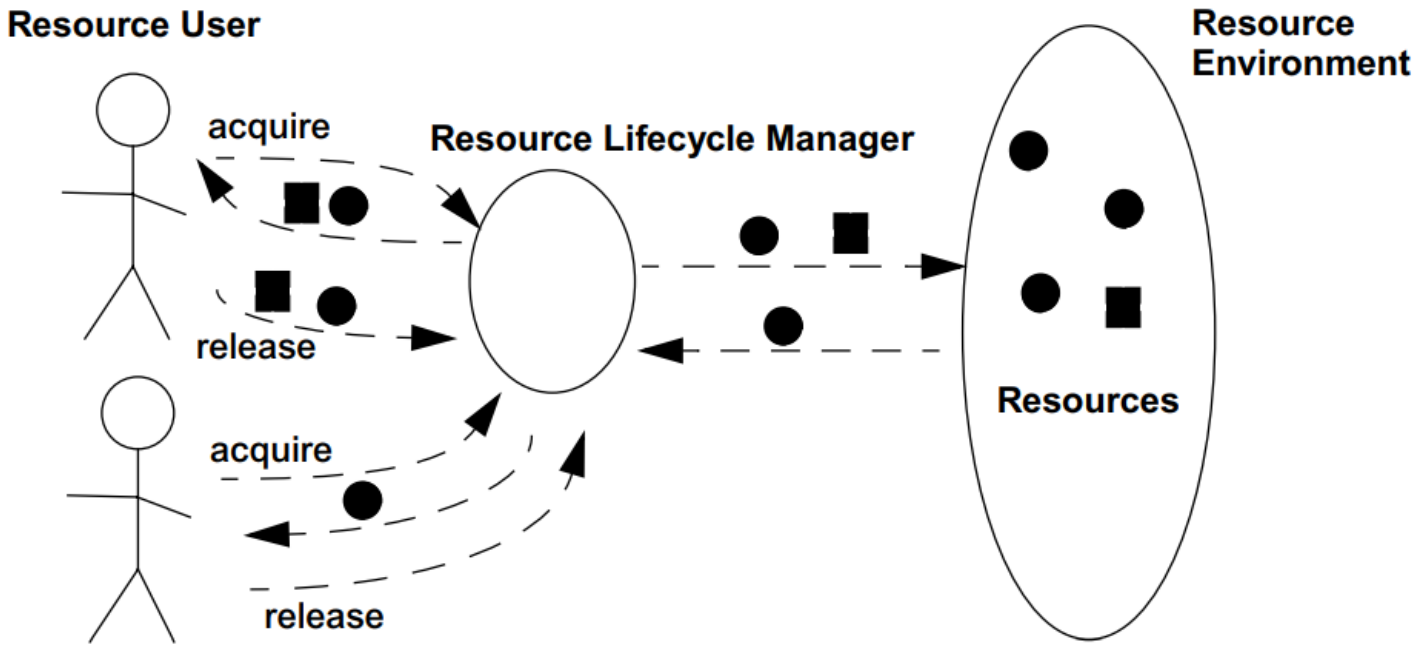

Abbildung 12: Resource Lifecycle Interaktion

Der Ablauf dieses Systems sieht dann folgendermassen aus:

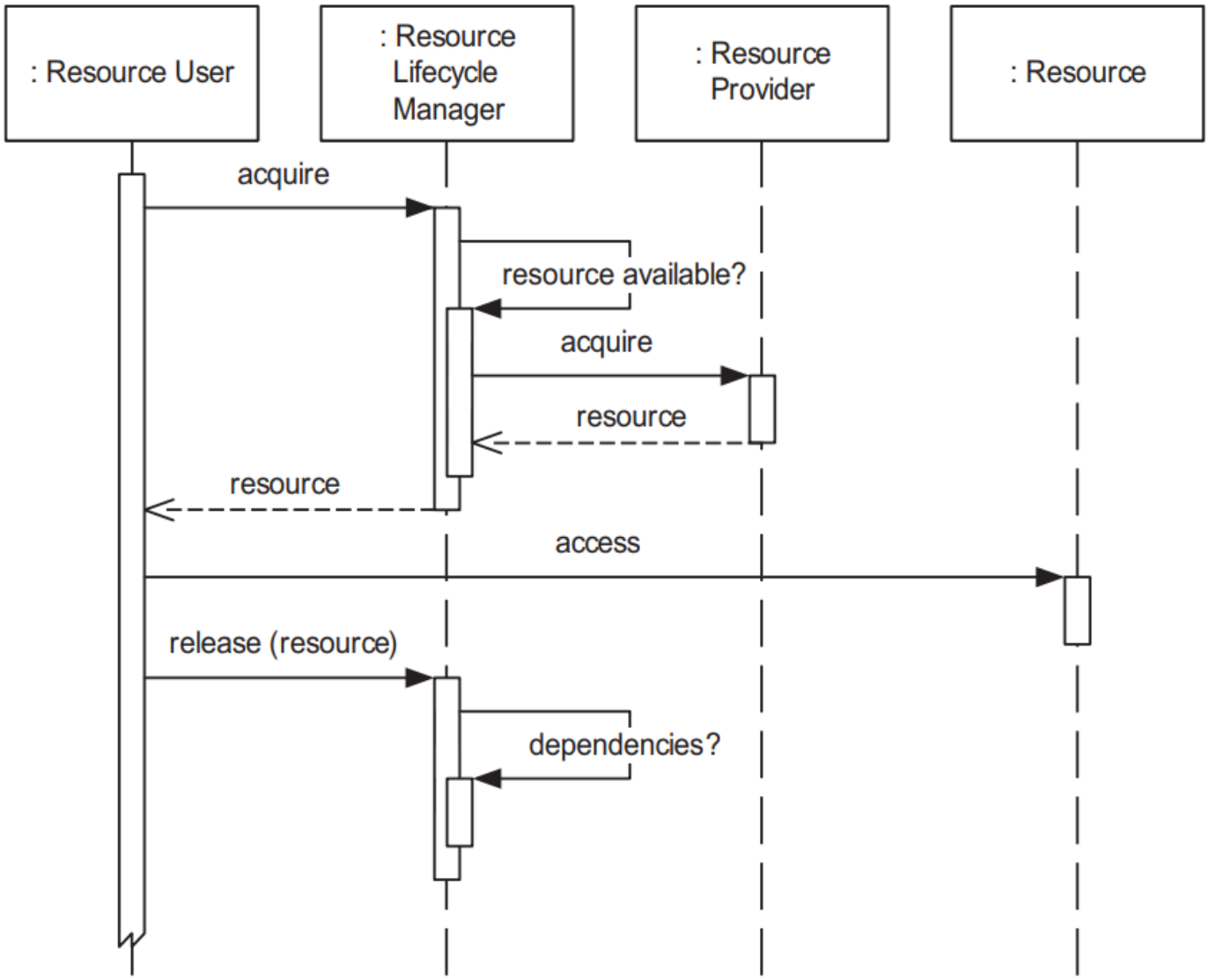

Abbildung 13: Resource Lifecycle Ablauf

Falls dieselbe Ressource mehrmals angefragt wird, kann der RLM auch Optimierungen wie Pooling oder Caching vornehmen.

6.5 Implementation

Zum Implementieren dieser Lösung braucht es 7 Schritte:

1. Ermitteln der zu verwaltenden Ressourcen
2. Definieren wie Ressourcen erstellt und Verwendung angefragt wird:

- Zum erstellen können z.B. das Factory Method Pattern oder das Abstract Factory Pattern verwendet werden.
- Zum Verwenden können z.B. das Eager Acquisition, das Lazy Acquisition und das Partial Acquisition Pattern verwendet werden.
- Falls Ressourcen knapp werden, kann der RLM Anfragen zurückweisen.
- Falls Ressourcen lazy erstellet werden muss beachtet werden, dass das Erstellen oder Verwenden von grossen Ressourcen lange dauern kann.

3. Definieren der Ressourcenverwaltungssemantik:

- Ressourcen müssen effizient und effektiv verwaltet werden.
- Pooling kann dafür verwendet werden eine fixe Anzahl Ressourcen ständig verfügbar zuhalten, wie z.B. Threads und Connections
- Caching kann verwendet werden, wenn die Identität und der Zustand der Ressource benötigt wird.

- Caching hilft die Ressourcenverwendung zu minimieren.

4. Ressourcen Abhängigkeiten verwalten

- Isolieren von verschiedenen Ressourcenlebenszyklen in separate RLM
- Optimierungen für voneinander Abhängige RLM sind schwer umzusetzen.
- Abhängige Ressourcen können zusammen gruppiert werden damit sie Zugriff auf denselben Kontext haben.

5. Definieren von Ressourcenfreigabe Semantiken:

- Wenn eine Ressource nicht mehr benötigt wird, sollte sie automatisch freigegeben werden.
- Dafür kann das Leasing oder das Evictor Pattern verwendet werden.
- Zusätzlich kann ein Garbage Collector verwendet werden, um unbenutzte Ressourcen zu identifizieren.

6. Definieren von Ressourcenzugriff Semantiken:

- Der Zugriff auf die Ressource sollte möglichst einfach sein, kann z.B. mit dem Lookup Pattern umgesetzt werden.

7. Konfigurationsstrategien implementieren:

- Für jeden der bisherigen Schritte sollte es möglich seine Strategien für die Umsetzung zu definieren.
- z.B.: teure Ressourcen werden erst geladen sobald sie benötig werden und günstige oft verwendete werden sofort beim Startup geladen.
- Reflection kann verwendet werden um die Strategie dem Environment anzupassen
- Falls Abhängigkeiten zwischen Ressourcen verwaltet werden müssen, kann das Coordinator Pattern verwendet werden.

6.6 Consequences

6.6.1 Vorteile

- Effizienz
- Skalierbarkeit
- Leistung
- Transparenz
- Stabilität
- Kontrolle

6.6.2 Nachteile

- Single point of failure: Ein Bug kann dazu führen, dass das grosse Teile des RLM nicht mehr funktionieren.
- Flexibilität: Falls Ressourcen eine sehr spezifisches Behandlung brauchen ist der RLM vielleicht zu unflexibel um das umzusetzen.

6.7 Wichtige Eigenschaften

- Transparenz: RLM ist gegenüber Ressourcenbenutzer transparent
- Anpassungsfähigkeit: RLM kann für verschiedene Bedürfnisse entsprechende Strategien zum Management der Ressourcen anwenden

6.8 Examples

Component Container Der Container verwaltet den Lebenszyklus von Applikationskomponenten, und den Lebenszyklus von Ressourcen welche von ebendiesen verwendet werden. Beispiele:

- J2EE Enterprise JavaBeans
- COM+

Remoting Middleware Middlewaretechnologien (i.e. CORBA, .NET Remoting) setzen RLMs auf verschiedenen Ebenen ein. Unter anderem werden Connections und Threads verwaltet.

Grid computing Grid computing behandelt das Teilen und Aggregieren von verteilten Ressourcen. Teilnehmende Maschinen müssen ihre Ressourcen verwalten können. RLMs werden ebenfalls verwendet um die Ressourcen im Grid zur Verfügung zu stellen.

6.9 Prüfungsfragen

1. Der Resource Lifecycle Manager verwendet einen Resource Provider zum akquirieren der Ressource. [Richtig|Falsch]
2. Abhängigkeiten von Resourcen haben einen Einfluss auf ihren Lifecycle. [Richtig|Falsch]

7 Microservice API Patterns

Moderator: Loris Keller

7.1 Language scope and organization

7.1.1 Language Foundations

Glossar

7.1.2 Motivation

Erforderliche Architekturentscheide beim Entwerfen von Schnittstellen:

- Message exchange pattern (request-response vs. one way)
- Message exchange format: JSON, XML, etc.
- **Data contract**

Die beschriebenen Patterns sind unabhängig vom gewählten Message exchange Format und Pattern und fokusieren sich auf den Aspekt des **Data Contracts**.

Für die beschriebenen Patterns und die entsprechenden Architekturentscheidungen werden folgende Aspekte berücksichtigt (**forces**):

- **Latency** (*Client*): Netzwerkverhalten und Rechenaufwand des Endpoints inklusive (un-)marshalling
- **Throughput** und **scalability** (*Provider*): Antwortzeiten sollten nicht langsamer werden wenn Last grösser wird
- **Learning effort** und **modifiability**: Aspekte der Wartbarkeit (z.B. Rückwartskompatibilität für parallele Entwicklung, flexibilität beim Deployment)
- **Security**: Aspekte der Datensicherheit wie *Confidentiality* und *Integrity*
- **Coupling**: Kopplung und Wissen zwischen *Provider* und *Client*

7.1.3 Language Organization

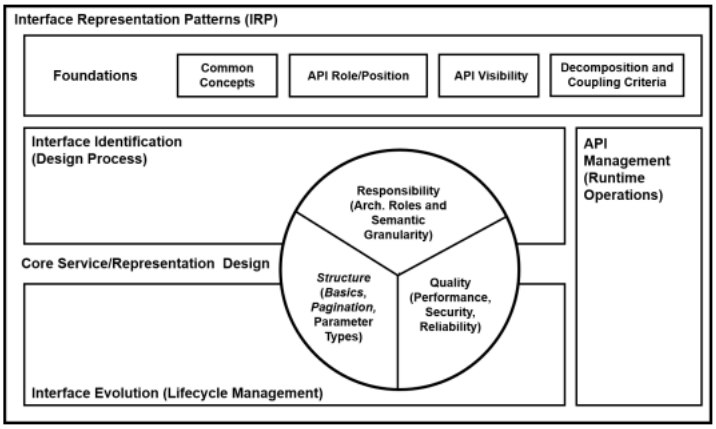

Abbildung 14: Pattern Language Categories

Zur Orentierung ist die Sprache in 7 Kategorien aufgeteilt:

- Interface **Responsibility**: Rolle der API Endpoints in der Architektur un deren Verantwortlichkeiten, *Wieso?*
- **Structure**: Wie viele Parameter sollen verwendet werden und wie werden diese strukturiert, *Was?*
- Delivery **Quality**: Sicherstellung der Qualität und effziente Nutzung der Resource, *Wie?*
- **Foundations**: Kontext der Sprache und Vokabular, Kriterien zu *Coupling*
- **Identification**: Prozess-Patterns zur Identification von API *Call* Kandidaten
- Service **Evolution**: Lifecycle management (Versionierung / Verwaltung)

- API **Management**: Distribution, Kontrolle und Analyze von APIs

7.2 Structural representation patterns

7.2.1 Context

Atomic Parameter, Atomic Parameter List, Parameter Tree, und *Parameter Forest* teilen den folgenden Context: Ein Provider will eine oder meherere Operationen seiner Clients mitels einem API Endpoint zur Verfügung stellen. Um die Kommunikation sicherzustellen, müssen sie einer Struktur für die Nachrichten zustimmen.

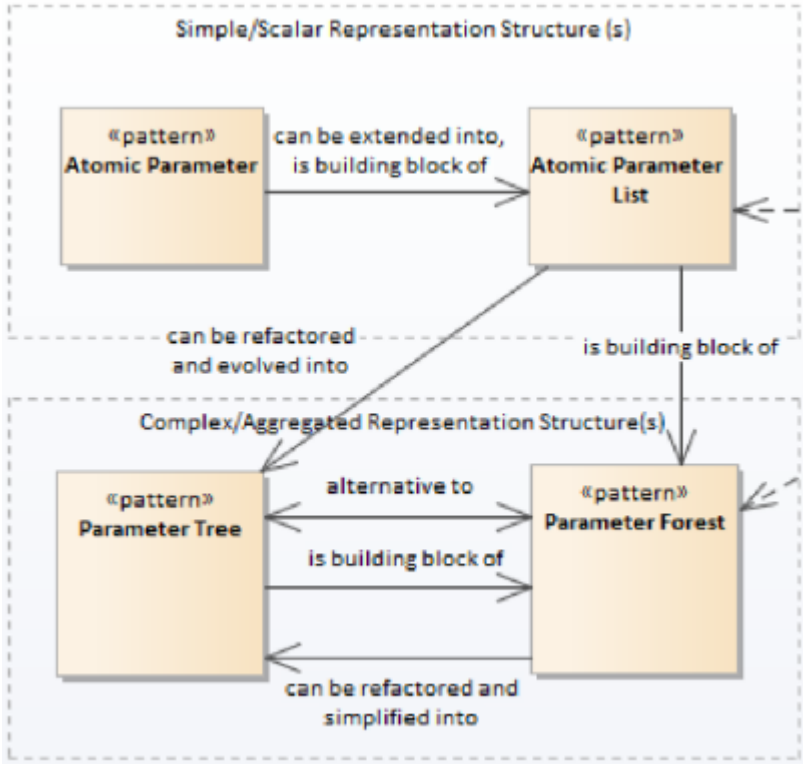

Abbildung 15: Structural representation patterns

7.2.2 Problem

Die `Structural representation pattern` beschreiben grundlegende Patterns die verschieden auf das gleiche übergeordnete Designanliegen beim erstellen der Schnittstelle reagieren. Dazu gehört das beantworten der Frage:

> Was ist eine angebrachte Anzahl von `in` und `out` paremeter eines API calls.

7.2.3 Forces

- Structure of the domain model and the system behavior and its impact on understandability
- Performance und resource use
- Loose coupling und interoperability
- Developer convenience und experience
- Maintainability
- Security und privacy

7.2.4 Atomic Parameter Pattern

7.2.4.1 Problem (Atomic Parameter)

Wie kann der API provider **einen** primitiven Datentypen als `in` oder `out` Parameter in der request / response messages definieren?

7.2.4.2 Forces (Atomic Parameter)

Oben erwähnte Strukturen der In-/Outputparameter sind essenziell in Service Contracts und teilen bereits beschriebene high-level forces.

Auch die Strukturen der request und response messages sind ein wichtiger Teil des technischen Service Contracts. Sie schaffen ein gemeinsames Verständnis zwischen dem Provider und Consumer.

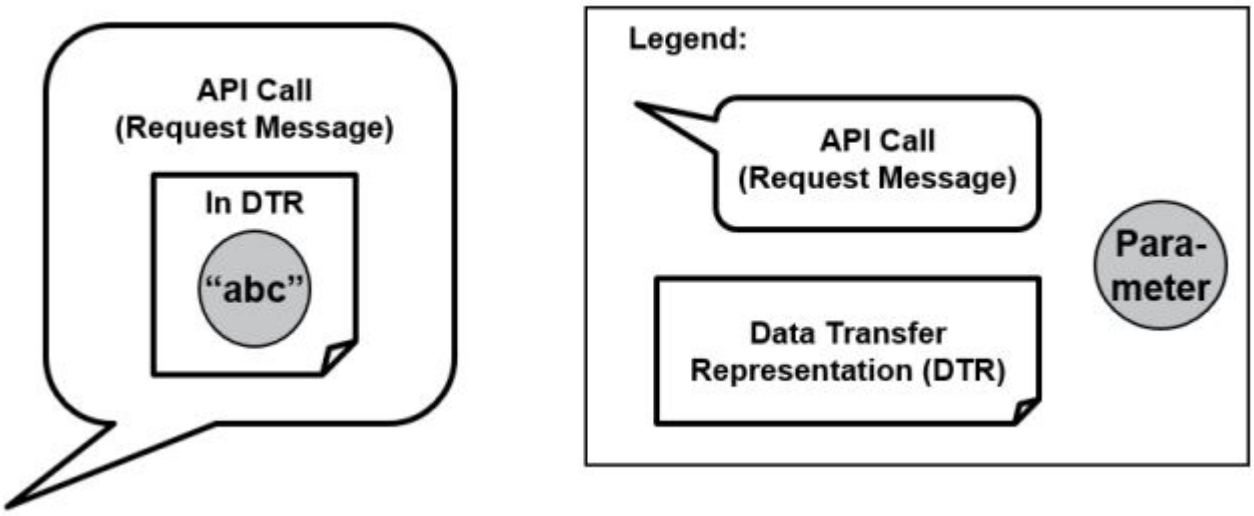

Abbildung 16: Atomic parameter pattern

Bei Unterspezifizierung des service contracts kann es zu Problemen der Interoperability führen wenn z.B. Optionalität realisiert wird. Bei Überspezifizierung wird es schwierig Rückwärtskompatibilität zu gewährleisten.

Hier kommen die simplen Datenstrukturen zum Zug. Manchmal werden nur minimale Daten benötigt um die für den Kommunikationspartner benötigten Informationen zu übertragen. Zum Beispiel beim Abfragen eines Status einer Ressource. Meistens ist es aber keine gute Idee Key-Value Paare auszutauschen, da dadurch der Test und Unterhaltsaufwand vergrössert wird und die Message dadurch unnötig aufgeblasen wird.

7.2.4.3 Solution (Atomic Parameter)

Man definiert einfach einen einzelnen Input- und Outputparameter im API Contract in Form von simplen, unstrukturierten Daten. Der Parameter bekommt einen Namen und einen Wertebereich. Es sollten immer wenn möglich primitive Datentypen gewählt werden, welche vom darunterliegenden Appliaktionsprotokoll unterstützt werden. Man sollte nur in Ausnahmefällen Typerweiterungen oder neue Typen als Parameter verwenden um den allgemeinen Test- und Unterhaltsaufwand zu minimieren.

Beispiel für einen RESTful HTTP Request welcher eine Instanz dieses Patterns implementiert:

```
@GET
@Path("/{claimId}")
    public ClaimDTO getClaimById(@PathParam("claimId") UUID claimId) {
    return claims.findById(claimId).map(ClaimDTO::create).orElseThrow(noSuchClaim);
}

curl http://localhost:8080/claims/a1e00494-e982-45f3-aab1-78a10ae3e3bd
```

7.2.4.4 Consequence (Atomic Parameter)

Bei Requests welche zusätzliche Informationen übermitteln sollen ist Atomic Parameter nicht geeignet, da sonst mehrere Requests abgesetzt werden müssten. Dazu greift man besser zu Atomic Parameter List.

7.2.4.5 Aktuelle Praxisbeispiele (Atomic Parameter)

Alle heute eingesetzen messagebasierten remote APIs verwenden dieses Pattern an verschiedenen Orten.

- Amazon Web Services (AWS) verwenden es um einen neuen Bucket zu erstellen mit einem `bucketName` als Parameter.
- Web API Spezifikationssprache Swagger hat die Notation von einem *parameter object* um einen skalaren Parameter zu definieren.

7.2.5 Atomic Parameter List Pattern

7.2.5.1 Problem (Atomic Parameter List)

Wie kann der API provider **mehrere** primitive Datentypen als `in` oder `out` Parameter in der request / response messages definieren?

7.2.5.2 Forces (Atomic Parameter List)

Die Datenstrukturen der request/repsone messages sind ein essenzieller Teil des API contracts. Die oben genannten Forces müssen ausbalanciert werden.

7.2.5.3 Solutions (Atomic Parameter List)

Für das Übermitteln von einem oder meherere einfachen, unstrukturierten Datentypen muss eine Liste von 'Atomic Parameters' (Number, string, or boolean values) definiert werden.

```
curl https://api.twitter.com/1.1/statuses/update.json?status=Hi%20there&lat=47.2266&lon=8.8184
```

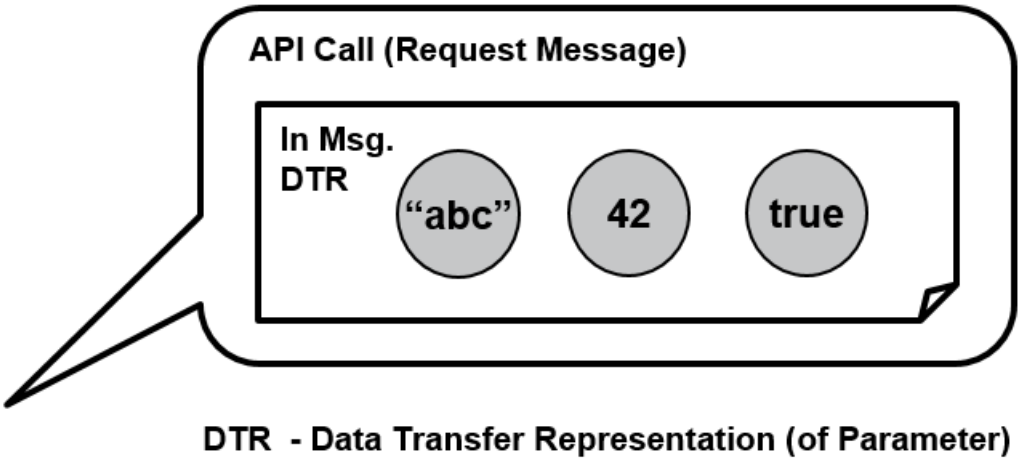

Abbildung 17: Atomic Parameter List

7.2.6 Consequences

Einige Integrationsplattformen verbieten den Kommunikationspartner mehrer Skalare in einem bestimmten Nachrichtentyp zu senden. Zum Beispiel erlauben viele Programmiersprachen nur einen `out` Parameter oder ein einzelnes Objekt als Rückgabewert.

7.3 Prüfungsfragen

1. Wenn eine Operation mehrere Informationen benötigt, müssten mit dem Atomic Parameter Pattern mehrere Requests/Calls ausgeführt werden?

   Ja, Atomic Parameters würden mehrere API calls benötigen.

2. Kann der Parameter im Atomic Parameter Pattern auch ein Objekt enthalten solange es nicht zu komplex ist?

   Nein, es dürfen nur skalare Werte verwendet werden.

8 Microservice API Patterns (continued)

Moderator: Leonard Obernhuber

8.1 Parameter Tree

8.1.1 Context (Parameter Tree)

Der Context wurde bereits auf der Ebene der structural representation patterns definiert.

8.1.2 Problem (Parameter Tree)

Wie kann der API provider **Tree Data Structures** in den Parameter einer Message definieren?
Wie kann der API provider **repetitive** oder **verschachtelte** Elemente in den Parameter einer Message definieren?

8.1.3 Forces (Parameter Tree)

Die Forces wurden bereits auf Ebene der structural representation patterns definiert.

8.1.4 Solution (Parameter Tree)

Die Basis der Parameterdarstellung ist ein einzelnes Element, welches wiederum eine oder mehrere untergeordnete, zusammengesetzte Datenstrukture(n) enthält.

`Tuple`, `Array` und `Trees` sind in den meisten Syntaxen verfügbar (Bsp. JSON-Objekte) und können als Basis dienen. Durch rekursive anwendung dieser Typen kann eine verschachtelt Strukur kreiert werden.

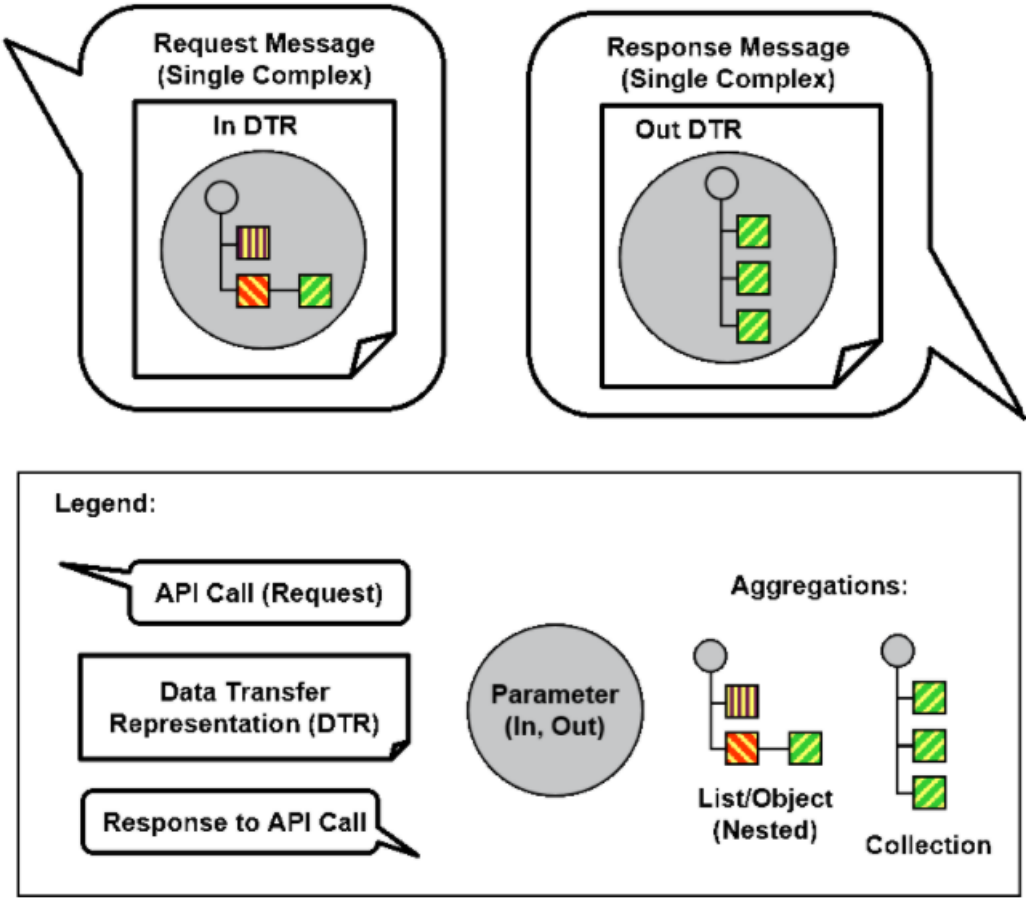

Abbildung 18: Parameter Tree pattern

Parameter Tree pattern mit zwei Variationen: Verschachtelte Liste, und homogene, flache Collection.

```
public class ClaimDTO {
    private final UUID id;
    private final String dateOfIncident;
    private final double amount;
    private final List<Link> links;
}

@PUT
@Path("/{claimId}")
public ClaimDTO updateClaim(@PathParam("claimId") UUID claimId,
                            @NotNull @Valid Claim claim) {
```

```
    boolean result = claims.update(claim);
    if (!result) {
        throw noSuchClaim.get();
    }
    return ClaimDTO.create(claim);
}
```

```
GET http://localhost:8000/claims/0afeb849-6d63-40b6-b52f-21dee16fdda5
```

```
{
  "claim": {
    "id": "0afeb849-6d63-40b6-b52f-21dee16fdda5",
    "dateOfIncident": "2021-10-12",
    "amount": 1024.0,
    "links": [
      {
        "uri": "http://localhost:8080/claims/0afeb849-6d63-40b6-b52f-21dee16fdda5",
        "title": null
      }
    ]
  }
}
```

8.1.5 Consequences

Wenn sich die Strukur des Domain Models natürlich als Tree abbilden lässt, ist dier Parameter Tree die optimale Variante.

8.1.5.1 Pros

- Wenn zusätzliche Daten (z.B. Sicherheitsinformationen) mit der Message übertragen werden müssen, kann der Parameter Tree die zusätzlichen Daten strukturell von den Domain-Daten trennen.
- Falls nötig, können zusätzliche Daten (z.B. Security Informationen) mit der Message übermittelt werden.
- Parameter Trees sind komplex und können unnötige Elementen enthalten wodurch Brandbreite verschwendet wird. Ist das Domain Model jedoch eine komplexe Baumstruktur, ist die Verarbeitung und auch die Bandbreitennutzung wesentlich effizienter, als das Verwenden von einfacheren Stukruren, die mehrer Messages brauchen.

8.1.5.2 Cons

- Zu grosse Komplexität: Es kann Sinnvoll sein, die Parameter Trees durch einfachere Strukuren zu ersetzen, wenn diese nicht wesentlich mehr Messages versenden.
- Die Komplexität kann zu Security und Privacy issues führen.
- Die Komplexität kann dazu führen, dass zu viele (strukturelle) Informationen geteilt werden -> Verletzung des *Loose coupeling* Prinzips.

8.1.6 Know Uses

- JIRA Cloud REST APIs benutzt dieses pattern in ihrer 'issue-createIssue' message.
- Twitter REST API benutzt dieses pattern in ihrem GET collections request. Die Response enthält ein einziges JSON-Object, dass untergeordnet z.B. alle Benutzer aufliestet.

8.2 Parameter Forest

8.2.1 Context (Parameter Forest)

Der Context wurde bereits auf der Ebene der structural representation patterns definiert.

8.2.2 Problem (Parameter Forest)

Wie tauscht man umfassende, sich wiederholende oder verschachtelte Daten in einer messagebasierten API zwischen Producer und Consumer aus?

8.2.3 Forces (Parameter Forest)

Die Forces wurden bereits auf Ebene der structural representation patterns definiert.

8.2.4 Solutions (Parameter Forest)

Jeder der Parameterstrukturen ist entweder ein Atomic Parameter, Atomic Parameter List, Parameter Tree oder ein weiterer Parameter Forest.

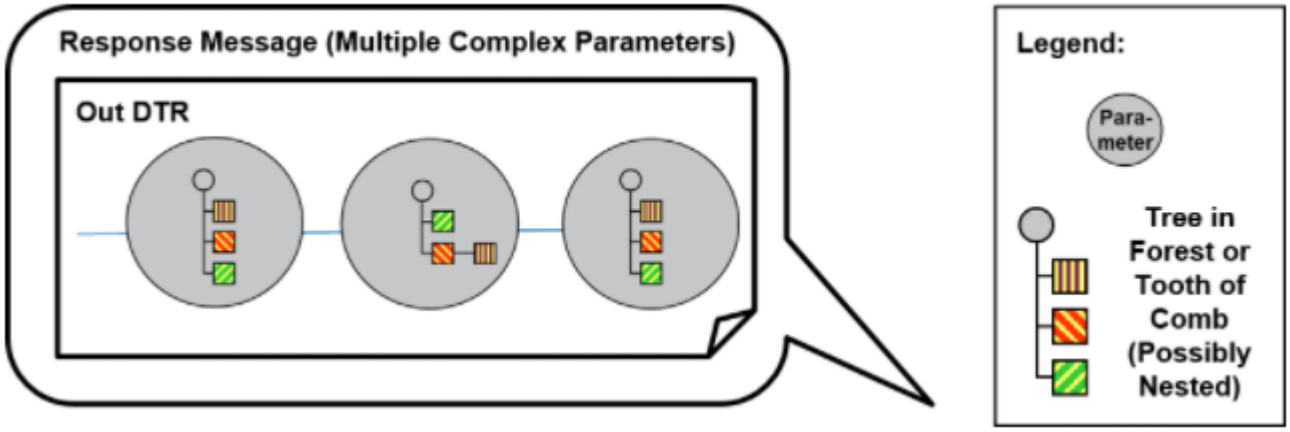

Abbildung 19: Parameter Forest pattern: message anatomy in iconic representation (here: response message consisting of three trees)

```
public interface IRPService {
    ResponseDTO forestInTreeOut(RequestDTO complexParameter1, AnotherRequestDTO complexParameter2);
}

public class RequestDTO {
  private float value;
  private String unit;

  [...]
}

public class AnotherRequestDTO {
  private int id;
  private NestedRequestDTO[] toothOfComb;

  [...]

public class ResponseDTO {
  private int id;
  private String dataField1;
  private String dataField2;

  [...]
}
```

8.2.5 Consequences (Parameter Forest)

8.2.5.1 Pro

- Wenn die Parameter eines Domänenmodells nicht einfacher dargestellt werden können als mit dem Parameter Forest Pattern, dann bietet es sich gegenüber den Alternativen Parameter List oder Parameter Tree mit einem künstlichen Root-Knoten an (wobei das dem Parameter Forest sehr ähnelt). Das sorgt für eine verständliche und schlanke Lösung.
- Wenn zusätzliche Daten mit dem Request mitgesendet werden müssen, bietet der Parameter Forest die Möglichkeit neben den komplexen Datenstrukturen zusätzliche Daten einfach zu übermitteln.
- Bei komplexen Strukturen wie z.B. einem Graphen ist die Übertragung und Verarbeitung sehr effizient.

8.2.5.2 Con

- Aufgrund der Komplexität eignen sich Parameter Forests hinsichtlich der Einfachkeit der Implementierung und Verständlichkeit für die Entwickler nicht immer. Oftmals kann man simplere Strukturen verwenden, wenn sie die Anzahl zu sendenden Requests nicht massgeblich erhöhen und die Komplexität verringern.
- Eine komplexe Verarbeitungslogik kann zu Sicherheitsproblemen führen, falls versehentlich schützenswerte Daten mitgesendet werden.
- Es ist verlockend unnötige Daten in den komplexen Datenstrukturen mitzusenden, was für unnötig aufgeblasene Requests und Responses führt.

8.2.6 Know Uses (Parameter Forest)

Die in der Twitter REST API verwendete Antwortstruktur verwendet ebenfalls einen Parameter Forest, wobei ein Baum ein Array von Objekten und ein zweiter Baum Kontrollinformationen und Metadaten wie Cursor und Page Tokens enthält.

8.3 Pagination

8.3.1 Context (Pagination)

In der Regel müssen Clients APIs anfragen, um Daten zur Darstellung für einen Nutzer oder zur weiteren Verarbeitung zu erhalten. Oft antwortet die API auf eine solche Anfrage mit einer grossen Anzahl von Daten, die eventuell sogar grösser ist als was der Client für die aktuelle Aufgabe braucht oder behandeln kann.

Daten können der gleichen Struktur entsprechen, typescherweise wenn Zeilen von einer relationalen Datenbank abgefragt werden. Sie können aber auch unterschiedliche Strukturen enthalten, zum Beispiel wenn sie von einer NoSQL Datenbank wie MongoDB abgefragt werden.

8.3.2 Problem (Pagination)

Wie können *heterogene* oder *homogene* Daten inkrementell an einen Client ausgeliefert werden, damit dieser nicht überfordert wird?

8.3.3 Forces (Pagination)

- Performance (latency, message processing), scalability, and resource use (bandwidth, memory, CPU power)
- Data set size and data access profile
- Loose coupling and interoperability
- Developer convenience and experience
- Security and data privacy
- Session awareness and isolation

8.3.4 Solutions (Pagination)

Grosse Datensets sollten in kleinere, verwaltbare Stücke (Pages) zerteilt werden. Pro Response sollte eine dieser Pages mit zusätzlichen Informationen wie “Totale Anzahl Einträge” oder “Anzahl verbleibende Pages” ausgeliefert werden. Zusätzlich sollte es dem Client möglich sein, den Resultatbereich selbst einzuschränken oder zu selektieren. Zu beachten ist auch, dass eine Reihenfolge für die Daten definiert werden muss.

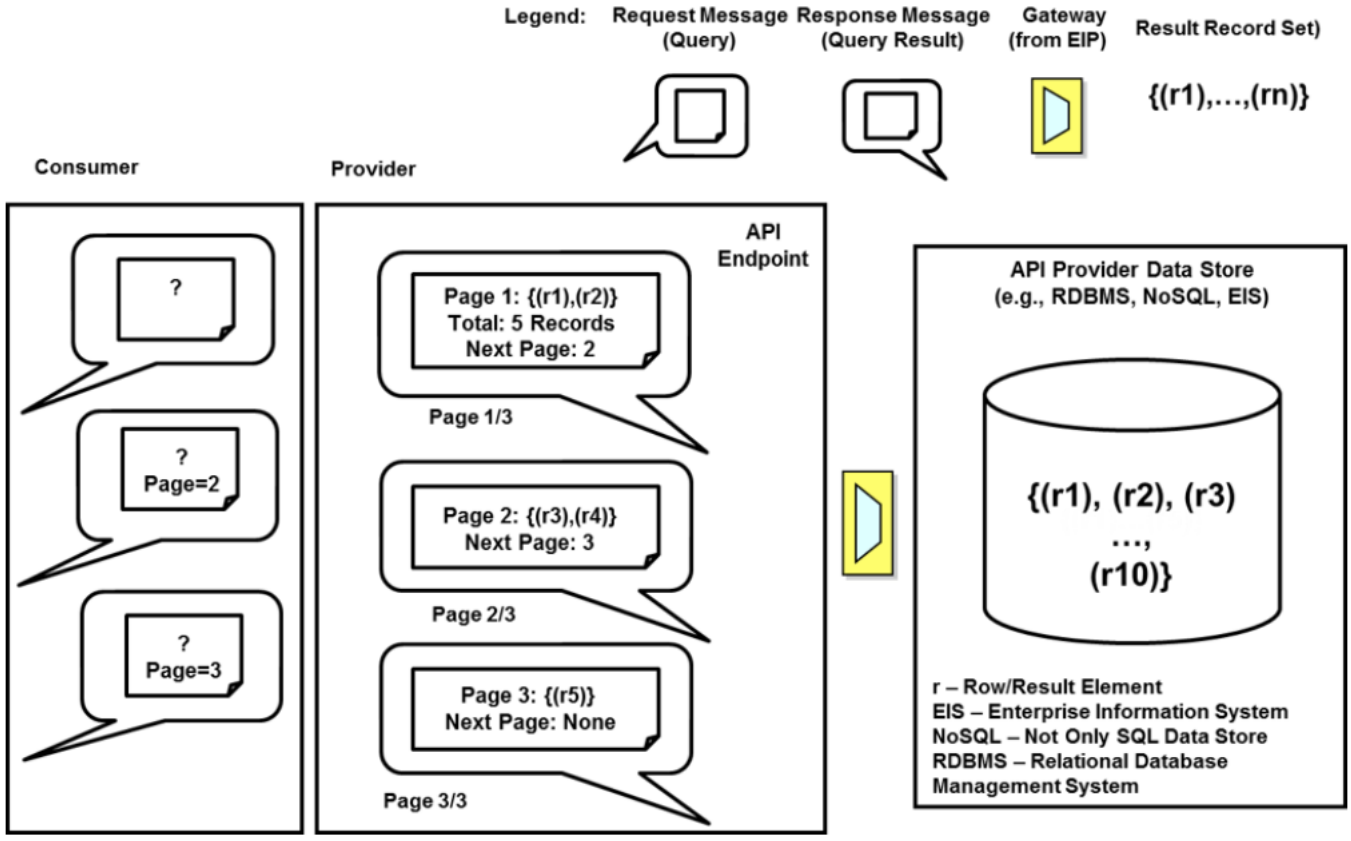

Abbildung 20: Pagination: query and follow-on request messages, response messages with filtered, partial result sets (pages)

```
interface PagedListResponse<T> {
  meta: {
    count: number;
    pageCount: number;
    totalItemCount: number;
    pageNumber: number;
    pageSize: number;
  };

  links: {
    previousPage: string;
    nextPage: string;
  };

  data: T[];
}
```

8.3.5 Varianten (Pagination)

Variante	Beschreibung
Paged-Based	Daten wird in gleichmässige Stücke (Pages) aufgeteilt. Eine einzelne Page wird über ihren Index identifiziert.
Offest-Based	Client definiert einen Offset der Angibt, wie viele Elemente vom Datenset übersprungen werden. Zusäzlich gibt er an, wie viele Elemente vom Set ausgwählt werden sollen. (Skip/Take)
Cursor-Based	Ist nicht abhängig von der absoluten Position eines Elementes im Set. Client gibt ein Wert zur Identifizierung eines Elements an und die Menge, die vom Set (von diesem Element aus) ausgewählt werden sollen.
Time-Based	Ähnlich zu *Cursor-Based* aber statt `Id` wird ein `Timestamp` benutzt. Wird seltener verwendet aber relevant für zeitbasierte Daten/Anzeigen.

8.3.6 Consequences (Pagination)

8.3.7 Pro

- Pagination verbessert den Verbauch von Ressourcen, sowie die Perfomance allgemein dramatisch. Dies ist dadurch bedingt, dass nur die Daten gesendet werden, welche zu einem bestimmten Zeitpunkt auch absolut nötig sind.
- Aus einem Sicherheitsstandpunkt, verhindert korrekte Pagination Denial-of-Service Attacken, die durch das übermässige Abfragen von Daten entstehen könnte.

8.3.8 Con

- Pagination ist nur dann anwendbar, wenn die Response sich an einem Datenset orientieren und in kleine Stücke (Pages) zerlegt werden können.
- Eine Response welche Pagination verwendet ist komplexer und sicher auch unangenehmer zu Benutzen als eine "einfache" Response.
- Pagination erhöht die Kopplung von Client und Provider. Dies kann aber durch ein konsistente Umsetzung der Pagination minimiert werden.
- Wenn ein Client nicht sequentiell auf Pages zugreiffen möchte, sondern mehr Kontrolle über die Selektion des Resultats haben möchte, müssen zusätzliche Parameter definiert und validiert werden.

8.3.9 Know Uses (Pagination)

Viele Verwendungen, gerade in der aktuellen Web-Entwicklung.

- Twitter API
- Confluence API
- StackExchange API

8.4 Prüfungsfragen

1. Ein Parameter Tree Pattern, das nur aus Atomic-Parametern besteht ist äquivalent zum Atomic Parameter List Pattern? *Ja*
2. Wird Pagination vor allem dazu verwendet, Datensets besser zu strukturieren? *Nein*

9 Interface Evolution Patterns

Moderator: Daniel Els

9.1 Version Identifier

9.1.1 Context

Produktiv eingesetzte APIs werden stetig weiterentwickelt. Versionen mit neuen Features erscheinen, welche ab einem bestimmten Zeitpunkt nicht mehr abwärtskompatibel sind.

9.1.2 Problem

Wie kann ein API Provider seine technischen Möglichkeiten und das Vorhandensein inkompatibler Änderungen an Clients spezifizieren um Fehlverhalten vorzubeugen?

9.1.3 Forces

- *Exakte Identifikation* der API Version
- Minimierung der durch API-Änderungen verursachten *Auswirkungen* auf den Client
- Gewährleistung, dass API-Änderungen nicht *versehentlich die Kompatibilität* zwischen Client und Provider auf semantischer Ebene *unterbrechen*
- *Rückverfolgbarkeit* der verwendeten API-Versionen zur *Governance*

9.1.4 Non-solution

Oft releasen Firmen ihre APIs ohne einen Plan zur Kontrolle über die Versionen zu haben weil sie denken das könne man im Nachhinein irgendwann machen. Das ist einer der Hauptfaktoren warum viele Projekte gescheitert sind.

9.1.5 Solutions

Man verwendet eine explizite Versionsangabe. Dieser Versionsidentifikator gehört in die API-Beschreibung und in die ausgetauschten Nachrichten. Oftmals verwendet man hierzu einen numerischen Wert um den Entwicklungsfortschritt und Reifegrad der Software anzugeben. Die Versionsangabe sollte nur an einem Ort in der API zugänglich gemacht werden um Konsistenzprobleme zu verhindern.

9.1.5.1 XML Namespaces

```
<soap:Envelope>
    <soap:Body>
        <ns:MyMessage xmlns:ns="http://www.nnn.org/ns/1.0/">
            ...
        </ns:MyMessage>
    </soap:Body>
</soap:Envelope>
```

9.1.5.2 JSON dediziertes Versionsattribut

```
{
    "version": "1.0",
    "products": [
        {
            "productId": "ABC123",
            "quantity": 5;
            "price": 5.00;
        }
    ]
}
```

9.1.5.3 Endpunkt URL

```
GET v2/customers/1234
```

9.1.5.4 API Domain Name

```
GET /customers/1234
Host: v2.api.service.com
```

9.1.5.5 HTTP content type header

```
GET /customers/1234
Accept: text/json+customer; version=1.0
```

9.1.6 Consequences

9.1.6.1 Pros

- Die Verwendung dieses Patterns hilft bei der Identifizierung von APIs und der klaren Kommunikation über API-Operationen und Messages.
- Verringert Probleme aufgrund von unerkannten semantischen Änderungen zwischen API-Versionen, die die Kompatibilität versehentlich unterbrechen.
- Ermöglicht die Nachverfolgung, welche Version der Nutzdaten von den Clients tatsächlich verwendet wird.

9.1.6.2 Cons

- Durch die Änderung des Versions-Identifiers müssen Clients möglicherweise auf eine neue API-Version aktualisieren, obwohl sich die Funktionalität, welche sie benutzen, garnicht geändert hat.

9.1.7 Know Uses

Die meisten Firmen, welche intern Software produzieren verwenden entweder intern oder extern ein solches Versionsidentifizierungsschema. Beispiele dafür sind Public Web APIs wie Facebook Graph API oder Twitter API.

9.2 Semantic Versioning

9.2.1 Context

Bei der Versionserkennung aus einer einzigen Zahl geht **nicht** Zwingend hervor, wie bedeutend eine Änderung zwischen verschiedenen Versionen ist. Die Clients möchten Auswirkungen schnellstmöglich erkennen, um Migrationen auf neue Version ohne grossen Aufwand machen zu können. Die Providers müssen verschiedene Versionen verwalten, um den Clients Garantieren zu können, dass breaking changes ihre Applikation nicht beeinträchtigen.

9.2.2 Problem

Wie kann ein Steakholder API-Versionen vergleichen und deren Kompatibilität erkennen?

9.2.3 Forces

- Minimaler Effort zum erkennen von *Inkompatibilitäten der Versionen*
- *Verwaltbarkeit* der API-Versionen und Steuerungsaufwand
- Klarheit über *Auswirkungen von Veränderungen*
- Klare *Trennung von Änderungen* mit unterschiedlichen Auswirkungen und unterschiedlicher Kompatibilität
- Klarheit beim *Zeitplan* für die Entwicklung der API.

9.2.4 Solutions

Hierarchisches Versionsschema mit drei Zahlen `x.y.z`, welches eine Abstufung der Änderungen ermöglicht. Meistens **Major-, Minor-** und **Patch-** Version genannt.

Versionsname	Alte Versionsnummer	Neue Versionsnummer	Veröffentlichung von...
Major	1.2.4	2.0.0	Inkompatiblen Änderungen *(breaking changes)*
Minor	1.2.4	1.3.0	Neuen, ergänzenden, kompatiblen Features
Patch	1.2.4	1.2.5	Bugfixes, Dokumentationsanpassungen

9.2.5 Consequences

9.2.5.1 Pros

- Grosse Klarheit bei der Darstellung von Auswirkung und Änderungen zweier API-Versionen auf deren Kompatibilität

9.2.5.2 Cons

- Erhöhter Aufwand bei der Ermittlung der Versionen, da es schwer sein kann, zu welcher Kategorie eine Änderung gehört

9.2.6 Know Uses

Das Pattern ist weit verbreitet und wird im Zusammenhang mit Remote-APIs und anderen Software-Artefakten verwendet.

Beispiel Azure DevOps Commits - Get

- `GET https://dev.azure.com/{organization}/{project}/_apis/git/repositories/{repositoryId}/commits?api-versi`
- `GET https://dev.azure.com/{organization}/{project}/_apis/git/repositories/{repositoryId}/commits?api-versi`
- `GET https://dev.azure.com/{organization}/{project}/_apis/git/repositories/{repositoryId}/commits?api-versi`

9.3 Two in Production

9.3.1 Context

Eine API verändert sich über die Zeit, wobei die Funktionalität erweitert, verbessert und verändert wird. Früher oder später sind die Änderungen einer neueren Version nicht mehr Rückwärtskompatibel mit älteren Versionen, was zu einem *Breaking Change* auf den Clients führt. Clients verschiedener APIs haben unterschiedliche *Life Cycles*, welche nicht zwingend mit denen der Provider übereinstimmen. Entsprechend kann ein Upgrade auf die neue API Version nicht forciert werden.

9.3.2 Problem

Wie kann ein Provider eine API schrittweise anpassen, ohne *Breaking Changes* beim Client auszulösen und gleichzeitig nicht eine grosse Anzahl von API Versionen pflegen zu müssen?

9.3.3 Forces

- Unterschiedliche *Life Cycles* zwichen Provider und Client ermöglichen, Provisonierung einer neuen API Version ohne *Breaking Change* bei Client
- Undetected backwards-compatibility problems bei neuen API Änderungen verhindern
- Roll back einer API Version
- Changes auf Client minimieren
- Verwaltungsaufwand für Clients einer älteren API Version minimieren

9.3.4 Solutions

Zwei oder mehrere sukzessive Versionen einer API bereitstellen, wobei die älteren Versionen entfernt oder deprecated werden. Dadurch ist sichergestellt, dass Clients zu einem späteren Zeitpunkt migrieren können.

- Variante zur Versionierung wählen Version Identifier Pattern.
- Anzahl an Versionen definieren (normalerweise Zwei) die parallel in Betrieb sind und Clients über *Life Cycle* Modell informieren.
- Wenn eine Version der API bereitgestellt wird, wird die älteste Version entfernt oder deprecated. Zusätzlich sollten alle Clients der älteren Version über Migrationsmöglichkeiten informiert werden. (Zusätzlich HTTP-Redirects)

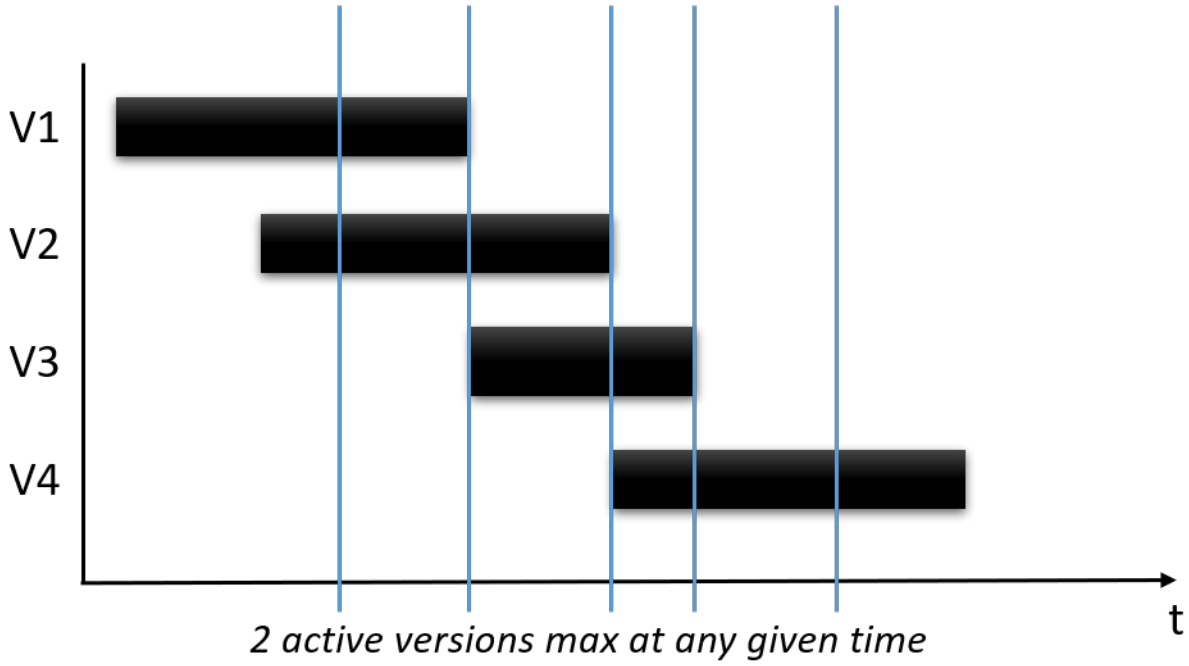

Abbildung 21: TwoInProduction

9.3.5 Consequences

9.3.5.1 Pros

- Clients können Änderungen planen und müssen nicht direkt migrieren, wenn eine neue API Version veröffentlicht wird
- Parallele Versionen haben einen hohen Kompatibilitätsgrad und erlauben einen Roll Back einer API Version
- Reduziert die Chance von undetected compatibility changes
- Reduziert Verwaltungsaufwand, da technische Änderungen ohne Rücksicht auf Rückwärtskompatibilität gemacht werden können

9.3.5.2 Cons

- Clients müssen *Breaking Changes* über die Zeit adaptieren
- Limitiert die Möglichkeit auf dringende Changes zu reagieren
- Aufwand/Kosten werden höher durch Verwaltung mehrerer API Versionen
- Zyklus der Änderungen an API gibt Migrationszeitraum vor

9.3.6 Known Uses

GitHub API

9.4 Limited Lifetime Guarantee

9.4.1 Context

Eine API Version wurde bereitgestellt und wird von mindestens einem Client benutzt. Der API Provider kann nicht beurteilen in welchen Zyklen die Clients angepasst werden und welcher Schaden entstehen würde, wenn man die Clients zu einem *Upgrade* zwingt. Der Provider möchte aber seine API weiterentwickeln, ohne *Breaking Changes* in der publizierten Version zu verursachen.

9.4.2 Problem

Wie kann ein Provider seinen Clients mitteilen, wie lange eine publizierte Version seiner API zur Verfügung steht?

9.4.3 Forces

- Anpassung des Clients aufgrund von Veränderungen an der API sollten planbar sein
- Verwaltungsaufwand für den Support von alten Clients minimieren

9.4.4 Solutions

Der API Provider garantiert keine *Breaking Changes* in einer publizierten Version für eine definierte Zeitdauer. Entsprechend muss ein Endzeiptunkt für jede API Version bezeichnet/bekanntgegeben werden.

Ein grosser Vorteil gegenüber Two in Production Pattern ist, dass keine weitere Koordination zwischen Provider und API erfolgen muss. Es ist bereits im Voraus bekannt, wie lange eine API Version unterstützt wird. Dieses Pattern richtet sich vor allem an die Stabilität des Clients, um ihn vor unerwarteten Änderungen zu schützen und Anpassungen planbar zu machen.

9.4.5 Consequences

9.4.5.1 Pros

- Gute Planbarkeit aufgrund der bereits bekannten Supportdauer

9.4.5.2 Cons

- Limitiert die Möglichkeit auf dringende Changes zu reagieren
- Zwingt Clients zu einem Upgrade an einem gewissen Zeitpunkt, welcher eventuell in Konflikt zu der Roadmap des Clients steht
- Keine Möglichkeit mit Clients umzugehen, die nicht mehr weiterentwickelt werden

9.4.6 Knwon Uses

Facebook Platform Versioning

9.5 Prüfungsfragen

1. Muss die Rückwärtskompatibilität gewährleistet werden wenn die Major-Version inkrementiert wird? *Nein*
2. Bei Limited Lifetime Guarantee ist ein wesentlicher Vorteil, dass keine Koordination mit den Clients nötig ist. Korrekt? *Ja*

10 Game Loop Pattern

Moderator: Loris Keller

10.1 Context

Früher waren Programme Batch-Orientiert, man hat sie gestartet und gewartet bis man ein Resultat erhält. Mit der Zeit hat man realisiert, dass es sehr ineffiezient ist ein Programm zu starten und Stunden später die Resultate abzuholen, besonders auch im Bezug auf die Erkennung und Behebung von Fehlern. Dies war die Entstehung von interaktiven Programmen, wobei Games mitunter die ersten dieser Art waren.

Die Grundlage von interaktiven Programmen sind User Inputs. Das Programm wartet auf einen Event vom User und behandelt diesen in sogenannten *Event Loops*.

```
while (true)
{
  Event* event = waitForEvent();
  dispatchEvent(event);
}
```

Im Unterschied zu anderer Software warten Games nicht explizit auf einen User Input. Sprich, dass Game läuft weiter auch wenn keine Eingabe erfolgt. Dies ist die Essenz eines echten Game Loops: Es verarbeitet User Input aber wartet nicht darauf.

```
while (true) {
  // Handles user input since last call
  processInput();

  // Advances game simulation (AI and physics)
  update();

  // Draws the game
  render();
}
```

10.2 Problem

Wie können User Inputs nicht blockierend verarbeitet und der Lauf der Zeit korrekt adaptiert werden?

10.3 Forces

- Keine Blockierungen
- Leistungsfähigkeit der Hardware
- Stromverbrauch
- Gameplay speed
- Entkopplung von Inputverarbeitung, Spiellogik und Rendering
- Koordination mit dem Event Loop der Plattform

10.4 Run, run as fast as you can (Fixed time step with no synchronization)

```
while (true)
{
    processInput();
    update();
    render();
}
```

Dies ist der simpelste und einfachste Game Loop, welcher aber nur einen kleinen Teil unserer Probleme löst.

10.4.1 Benefits

- Einfache implementierung
- User Inputs werden nicht blockierend verarbeitet
- Funktioniert gut, wenn spezifische Hardware vorausgesetzt ist und Stromverbauch nicht von Relevanz ist

10.4.2 Problems

- Keine Kontrolle wie schnell das Spiel läuft
- Auf schneller Hardware würde man nicht sehen was passiert

- Content-heavy oder Teile mit mehr AI oder Physics würden auf minderer Hardware real langsamer laufen

10.5 Take a little nap (Fixed time step with synchronization)

Das Problem, wie schnell ein Spiel laufen darf, kann mit einer fix definierten Zeitperiode für ein `sleep()` gelöst werden..

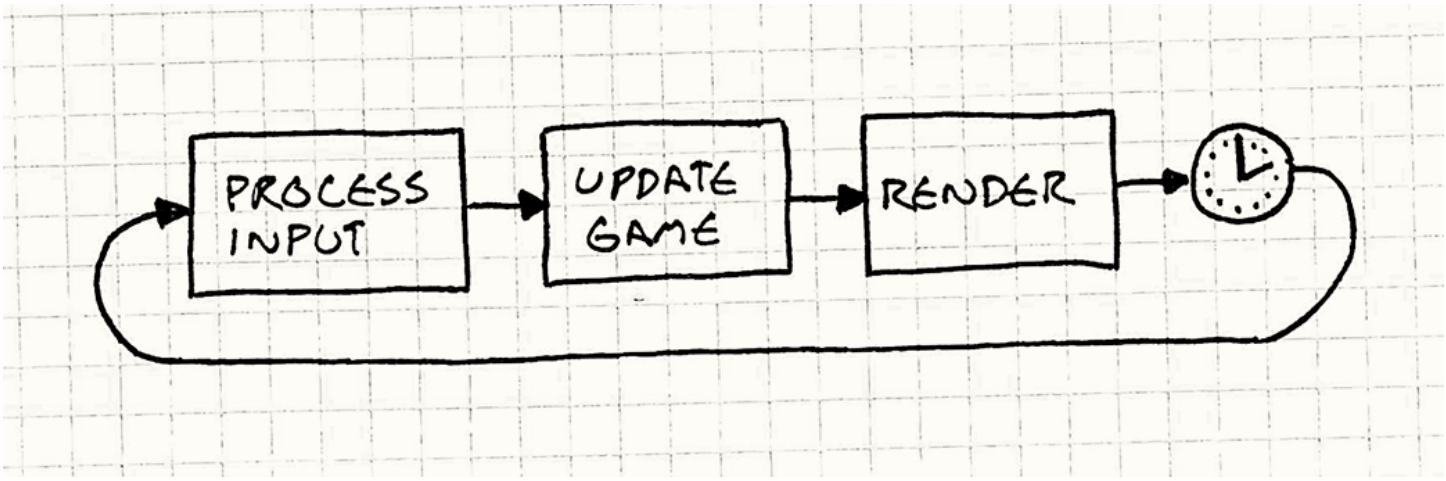

Abbildung 22: Game Loop Simple

```
while (true)
{
    var start = getCurrentTime();

    processInput();
    update();
    render();

    // Sleep to make sure the game does not run too fast!
    sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

10.5.1 Benefits

- Immer noch sehr einfache Implementierung
- Schonend für Stromverbrauch
- Spiel läuft nicht zu schnell

10.5.2 Problems

- Spiel kann zu langsam laufen
- Wenn `update` und `render` zu lange brauchen, wird gameplay verlangsamt

10.6 One small step, one giant step (Variable time step)

- Mit jedem `update` wird die **game time** um einen bestimmte Zeit vorwärts getrieben.
- Es braucht eine bestimmte **real time** um das zu verarbeiten.

Wähle ein variable time step , die von der **real time** abhängig ist. Dauert das verarbeiten länger, desto grössere Schritte macht das Spiel. Somit hält das Game immer mit der realen Zeit mit.

```
var lastTime = getCurrentTime();
while (true)
{
    var current = getCurrentTime();
    // Calculate the real time since last update()
    var elapsed = current - lastTime;
```

```
    processInput();
    // Update is responsible for advancing the game forward by the elapsed time.
    update(elapsed);
    render();
    lastTime = current;
}
```

Beispiel: Bei einem fixen time step bewegt sich ein Gegenstand entsprechend ihrer Geschwindigkeit über den Screen. Bei der variable time step skaliert die Geschwindigkeit mit der verstrichenen **real time** und bewegt sich weiter, je grösser dieser Zeitabschnitt ist. D.h egal ob mit 20 kleinen, schnellen oder vier grossen, langsamen Schritten, der Gegenstand benötigt die gleiche Zeit, um sich über den Screen zu bewegen.

10.6.1 Benefits

- Das Spiel läuft auf unterschiedlicher Hardware mit gleichbleibender Geschwindigkeit
- Spieler mit schnelleren Rechnern werden mit einem flüssigeren Spielablauf belohnt

10.6.2 Problems

- Nicht mehr Deterministisch
  - Auf unterschiedlichen Hardware werden Berechnungen unterschiedlich oft durchgeführt (floating point operations)

10.7 Play catch up (Fixed update time step, variable rendering)

Ein Teil der Engine welcher oft nicht an die Zeitvariable gekoppelt ist is das Rendering. Es wiedergibt nur einen Zeitpunkt und es ist daher egal wieviel Zeit seit dem letzten Rendering vergangen ist. Das kann man zu seinem Vorteil nutzen und einfach fixe Zeitschritte definieren. Das macht die Angelegenheit um einiges einfacher und stabiler.

Man nimmt die Echtzeitperiode zwischen den Ausfürungen des Game Loops. Daraus wird definiert wie viel Gamezeit simuliert werden muss für die Synchronisierung zwischen Echt- und Gamezeit. Das macht man mit fixen Zeitintervallen:

```
double previous = getCurrentTime();
double lag = 0.0;
while (true)
{
  double current = getCurrentTime();
  double elapsed = current - previous;
  previous = current;
  lag += elapsed;

  processInput();

  while (lag >= MS_PER_UPDATE)
  {
    update();
    lag -= MS_PER_UPDATE;
  }

  render();
}
```

Anfangs jedes Frames wird der `lag` anhand der verstrichenen Echtzeit definiert. Daraus wird errechnet wieviel die Gamezeit hinter der Echtzeit hinterherhinkt. Die Update-Aktion wird isoliert und so lange ausgeführt bis die Echt- und Gamezeit wieder synchron laufen und dann wird gerendert.

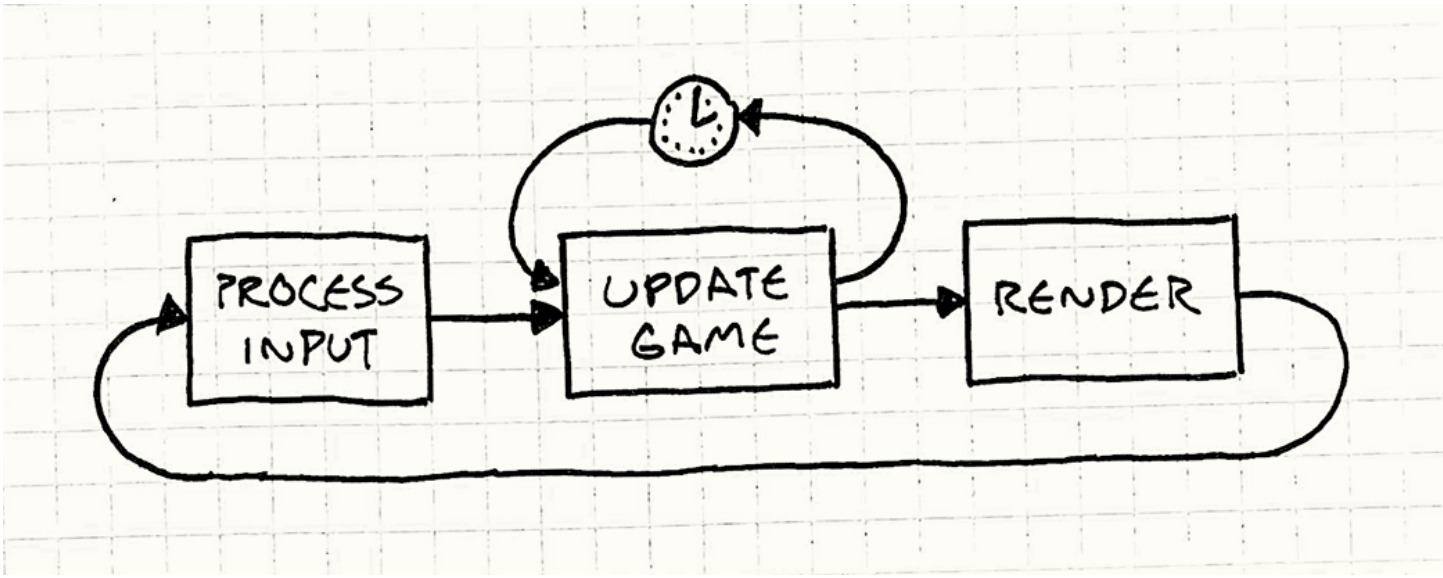

Abbildung 23: game-loop-fixed

Der Zeitintervall sagt hier aber nichts mehr über die *sichtbare* Framerate aus. `MS_PER_UPDATE` beschreibt nur die Granularität in der das Spiel aktualisiert wird. Bei längeren Intervallen wird die Rechenzeit grösser und die Framerate kleiner.

10.7.1 Benefits

- Gameplay speed ist Konsistent
- Updates in fixen Zeitintervallen
- Entkopplung von `update` und `render`

10.7.2 Problems

- Komplexität
- Update time step so klein wie möglich für high-end und genug gross für low-end

10.7.3 Stuck in the middle

Bei dem vorherigen Ansatz wird zwar in fixen Zeitintervallen aktualisiert, aber an zufälligen Zeitpunkten gerendert. Das führt dazu, dass die Anzeige oft Zeitpunkte zwischen Updates darstellt.

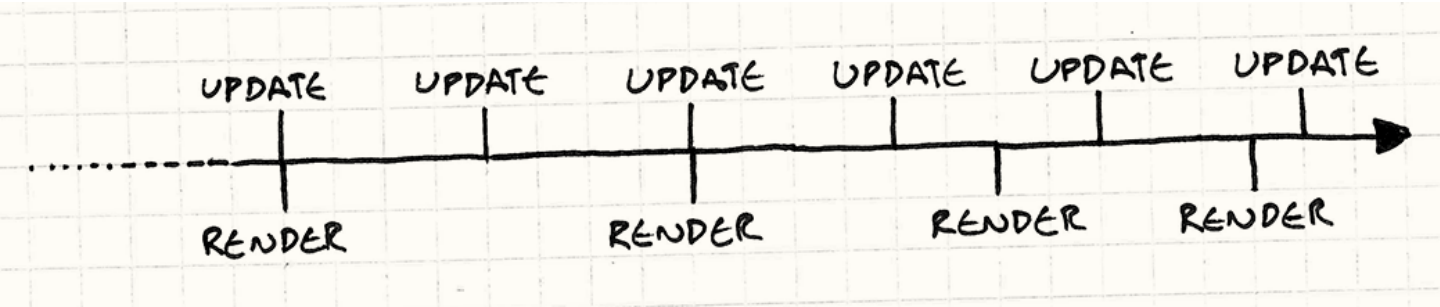

Abbildung 24: game-loop-timeline

Beispiel: Ein Gegenstand fliegt von einer Seite des Bildschirms zur anderen. Zwei Updates berechnen Start und Endpunkt. In Echtzeit wäre der Gegenstand in der Mitte des Screens, das Rendering zeigt aber nur den Startpunkt links. Der `lag` sagt genau aus wo sich die Kugel befindet und man kann die Position berechnen. Dazu gibt man einfach folgende Rechnung ins Rendering mit und lässt die Position berechen:

```
render(lag / MS_PER_UPDATE);
```

Diese sogenannte Interpolation kann aber in bestimmten Situationen falsch sein falls das Objekt durch etwas abgebremst wird. Das macht sich aber in den meisten Fällen nur minimal bemerkbar und ist gegenüber der flüssigen Darstellung zu vernachlässigen.

10.8 Design Decisions

10.8.1 Game Loop Owner

- **Platform:** Keine Entwicklung oder Optimierungen nötig, gute Plattformintergation, wenig Kontrolle
- **Game Engine:** Keine Entwicklung oder Optimierungen nötig, wenig Kontrolle
- **Write it yourself:** Totale Kontrolle, Integration mit Plattform muss selbst erarbeitet werden

10.8.2 Power Consumption

- **Run as fast as it can:** Desktop-PCs, Überschüssige Cycles werden zur erhöhung von FPS oder verbesserung von Graphik verwendet
- **Frame Rate limitieren:** Mobile Games, Sleep wenn Berechnungen abgeschlossen sind

10.9 Prüfungsfragen

1. Mit der Variante variable time step läuft das Spiel auf unterschiedlicher Hardware mit gleichbleibender Geschwindigkeit und eignet sich somit für multiplayer Spiele?
   Nein
2. Wird ein einfaches `sleep` in den Gaming Loop eingebaut, so wird Gameplayqualität von langsamen Maschinen beeinflusst?
   Nein

11 Component Game Pattern

Moderator: Leonard Obernhuber

11.1 Context

Moderne Computerspiele haben diverse Aufgaben wie Inputverarbeitung, Physik-/Kollisionsberechnungen, Rendering und Soundwiedergabe. Dabei können schnell unübersichtliche und nicht wartbare Klassen entstehen, die zu viele unterschiedliche Aufgaben haben.

Beispiel der Nichteinhaltung von *separation of concern*:

```
if (collidingWithFloor() && getRenderState() != INVISIBLE) {
  playSound(HIT_FLOOR);
}
```

Eine starke Kopplung kann insbesondere bei der Verwendung von Multi-Core Spielen zu Concurrency Problemen führen. Daher sollte man die Aufgabenbereiche klar definieren und in Komponenten auslagern. Das sorgt für etwas mehr Komplexität, aber dafür auch für weniger Fehleranfälligkeit, Entkopplung und Wiederverwendbarkeit des Codes.

11.2 Problem

Wie ermöglicht man, dass eine einzelne Entität mehrere Domänen umfassen kann, ohne dass die Domänen aneinander gekoppelt sind?

11.3 Forces

- Entkopplung der Domänen (Separation of Concerns)
- Wiederverwendbarkeit der Komponenten
- Wartbarkeit des Codes
- Multi-Threading

11.4 Solution

11.4.1 Bjorn the problematic baker

Hier dargestellt ist die problematische Bjorn-Klasse. Aktuell wäre die Klasse noch gut verwaltbar, jedoch wäre in einem realen Spiel die Logik viel komplexer und wir müssten beispielsweise ebenfalls Sound-Effekte für Bjorn implementieren.

Die Logik der aktuellen `update` Methode hat nichts mit dem Pattern zu tun, sondern dient nur zur Illustration. Wichtig ist zu erkennen, das wir momentan 3 verschiedene *Domains* verarbeiten: User Input, Physik und Rendering.

```
const WALK_ACCELERATION: number = 1;
class Bjorn {
  private positon: Position;
  private volume: Volume;
  private velocity: number = 0;
  private spriteStand: Sprite;
  private spriteWalkLeft: Sprite;
  private spriteWalkRight: Sprite;

  update(world: World, graphics: Graphics): void {
    // Apply user input to hero's velocity.
    switch (Controller.getJoystickDirection()) {
      case Direction.Left:
        this.velocity -= WALK_ACCELERATION;
        break;

      case Direction.Right:
        this.velocity += WALK_ACCELERATION;
        break;
    }

    // Modify position by velocity.
    this.position.x += velocity;
    world.resolveCollision(this.volume, this.position, this.velocity);

    // Draw the appropriate sprite.
    sprite = this.spriteStand;
    if (velocity < 0) {
      sprite = this.spriteWalkLeft;
    } else if (velocity > 0) {
      sprite = this.spriteWalkRight;
    }

    graphics.draw(sprite, this.position);
  }
}
```

11.4.2 Components

Wir erreichen eine Entkopplung der verschiedenen Domains, indem wir diese in verschiedene Komponenten aufteilen. Wir haben eine Komponente welche Input vom Benutzer verarbeitet, sowie spezifische Bjorn-Komponenten welche sich um die Physik und das Rendering von Bjorn kümmern. Die `DemoInputComponent` zeigt ebenfalls einen grossen Vorteil dieses Patterns, da wir für einen "Demo-Modus" den Input des realen Spielers durch einen Computergesteuerten Input ersetzen können.

```
class PlayerInputComponent implements InputComponent {
  constructor(readonly walkAcceleration: number = 1) {}

  update(gameObject: GameObject): void {
    switch (Controller.getJoystickDirection()) {
      case Direction.Left:
        gameObject.velocity -= this.walkAcceleration;
        break;
```

```
      case Direction.Right:
        gameObject.velocity += this.walkAcceleration;
        break;
    }
  }
}

class DemoInputComponent implements InputComponent {
  update(gameObject: GameObject): void {
    /* Demo functionality */
  }
}

class BjornPhysicsComponent implements PhysicsComponent {
  update(gameObject: GameObject, world: World): void {
    /* Much physics, such gravity */
  }
}

class BjornGraphicsComponent implements GraphicsComponent {
  update(gameObject: GameObject, graphics: Graphics): void {
    /* Draw all the things */
  }
}
```

11.4.3 Transformed Bjorn

Nach der Transformation der Bjorn-Klasse enthält die Klasse keine spezifische Logik, welche nur für Bjorn gelten würde. Sie dient als Container und kann entsprechend auch generisch und mit unterschiedlichen Komponenten zusammengesetzt werden. Aufgrund dessen können wir auch sagen, dass dieser Container ein generisches `GameObject` darstellt.

```
class GameObject {
  velocity: number = 0;
  position: Position;

  constructor(
    private readonly input: InputComponent,
    private readonly physics: PhysicsComponent,
    private readonly graphics: GraphicsComponent
  ) {}

  update(world: World, graphics: Graphics): void {
    this.input.update(this);
    this.physics.update(this, world);
    this.input.update(this, graphics);
  }

  static createBjorn(): GameObject {
    return new GameObject(
      new PlayerInputComponent(),
      new BjornPhysicsComponent(),
      new BjornGraphicsComponent()
    );
  }
}
```

11.5 Consequences

11.5.1 Pros

- Bessere Wiederverwendbarkeit von Logik der einzelnen *Domains*

- Reduziert Kopplung der einzelnen *Domains* massiv
- Austauschbarkeit der Komponenten

11.5.2 Cons

- Je nach Granularität der Komponenten nimmt die Komplexität enorm zu
- Korrekte Kommunikation zwischen den Komponenten ist anspruchsvoll
- Golden Hammer

11.6 Design Decisions

Die wichtigste Design Frage, die es zu beantworten gilt, ist: "**Welche Sets von Components werden gebraucht?**"

-> Je grösser und komplexer die Engine, desto feiner die Components.

Es gibt nicht die absolute Lösung und meistens werden auch mehrere Designs angewendet.

11.6.1 Wie kommt das Objekt zu seinen Components¿

11.6.1.1 Das Objekt erstellt seine eigenen Components

- Es ist sichergestellt, dass immer **alle** Components erstellt werden.
- *High Coupling:* Das Objekt kann nicht ohne weiters neu konfiguriert werden.

11.6.1.2 Das Objekt wird ausserhalb des Components erstellt.

- Mehr flexibilität, da das Verhalten des Objekts durch die übergabe von verschiedenen Components komplett verändert werdern kann.
- *Low Coupling.*

11.6.2 Wie kommunizieren die Components untereinander?

Perfekt entkoppelte Components, die isoliert funktionieren, sind ein schönes Ideal, das sich in der Praxis aber nicht wirklich umsetzten lässt.

11.6.2.1 Durch das Ändern des Objektstates

- Components sind weiterhin decoupled.
- Alle Informationen müssen in das Container-Objekt übertragen werden.
  - Nicht immer relevant für alle Components
  - Zu viele Component-Konfigurationen können Speicher verschwenden
- Die Kommunikation wird implizt und abhängig von der Reihenfolge

11.6.2.2 Durch direkte kommunikation untereinander

Alle Components, die miteinaner kommunizieren müssen, erhalten eine direkte Referenz zueinander.

- Einfach und schnell, die Kommunikation muss nicht über das Container-Objekt.
  - *Free-for-all*: Es können alle `public` methoden aufgerufen werden.
- Die beiden Components sind tightly coupled.

```
class BjornGraphicsComponent implements GraphicsComponent {
  constructor(private readonly physics: BjornPhysicsComponent) {}

  update(gameObject: GameObject, graphics: Graphics): void {
    /* Much physics, such gravity */
    if (this.physics.isOnGround()) {
      /* do ground action */
    } else {
      /* do some flips */
    }
  }
}
```

11.6.2.3 Via dem Versenden von Messages

Die komplexeste Alternative ist das bauen eines Nachrichtensystems im Container-Objekt, damit die Components nachrichten untereinander austauschen können.

- *Sibling components are decoupled.*
- Container-Objekt ist weiterhin einfach.

```
// Base component that will be implemented by all other components
class Component {
  public abstract receive(message: number): void;
}

class ContainerObject {
  private readonly components: Component[];

  // Send the messsage to each component
  public send(message: number): void {
    for (let component of this.components) {
      component.receive(message);
    }
  }
}
```

11.7 Aktuelle Praxisbeispiele

- `Unity` framework's core `GameObject`
- `Delta3D` engine base `GameActor`
- `Microsoft's XNA` game framework core `Game` class.

11.8 `Component`-Pattern vs `Strategy`-Pattern

Dieses Pattern hat viele Ähnlichkeiten mit dem *GoF* `Strategy`-Pattern. Beide Patterns kümmern sich um das Verhalten eines Objekts und das delegieren zu Sub-Objekten.

- Das `Strategy`-Pattern ist jedoch meistens **Stateless** -> Es kapselt einen Algorithmus, aber keine Daten.
- Components sind etwas selbständiger, da sie meist auch einen eigenen State haben.

11.9 Prüfungsfragen

1. Sollte man für die Kommunikation zwischen Komponenten nur eine Variante verwenden? **Nein**
2. Mit Hilfe dieses Patterns kann ein Auto-Pilot für einen Spieler auf einfache Art und Weise implementiert werden? **Ja**

12 Event Queue

Moderator: Daniel Els

12.1 Intro

Fast jedem sind die Ausdrücke *Event Queue*, *Message Queue* oder auch *Event Loop* geläufig. Typescherweise finden sich solche *Event Loops* vor allem bei der GUI Programmierung. Für jeden Klick eines Buttons oder Tastenanschlag generiert das Betriebssystem einen Event, welcher von der Applikation behandelt werden muss. Sprich, die Applikationen wartet auf die Eingabe von User Inputs.

Wie wir bereits gelernt haben, warten Spiele nicht auf die Eingabe von Benutzern, sondern arbeiten diese nach und nach in Form von einem *Game Loop* ab. Spiele verwenden aber oft eine *Event Queue* zur Kommunikation der Systeme oder *Komponenten*. Beispielsweise will man eher nicht eine Logik zur Anzeige von Tutorials innerhalb der "Combat Logik" implementieren.

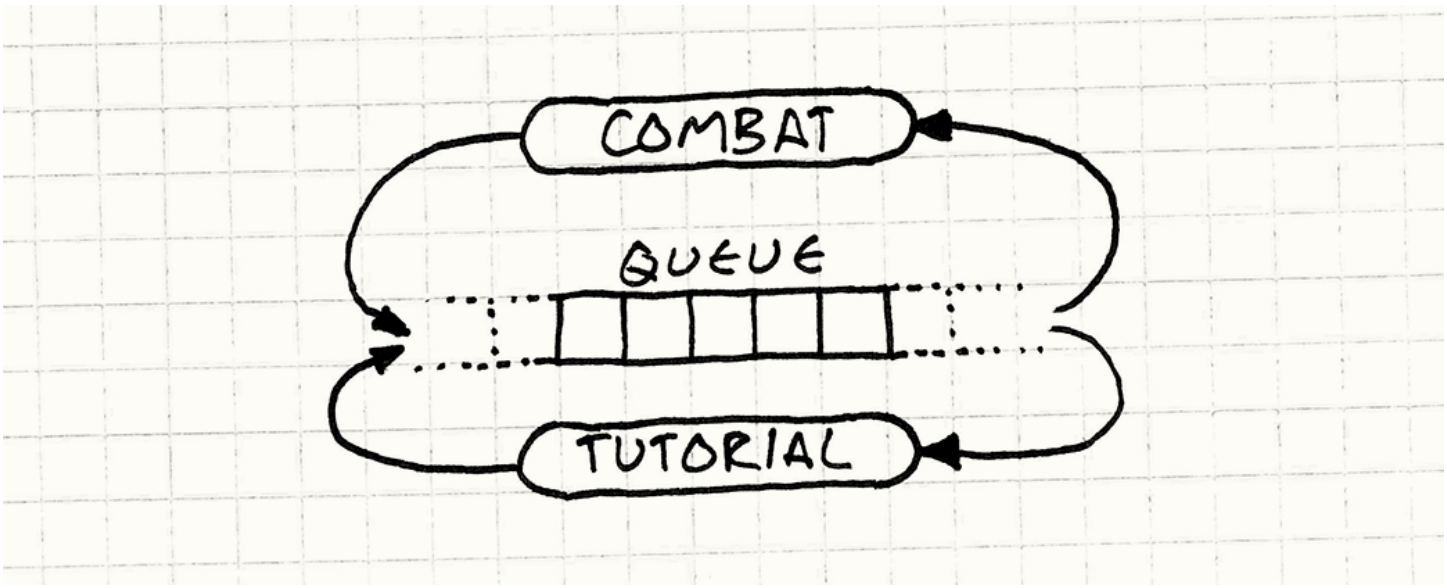

Abbildung 25: event-queue-central

12.2 Problem

Wie kann die Verarbeitung vom Auftreten eines Events oder von der Anfrage eines Requests entkoppelt werden?

12.3 Forces

- Kommunikation zwischen oder innerhalb von Komponenten / Domains
- Keine blockierenden "Aufrufe"
- Entkopplung zwischen dem Senden eines *Requests* oder eines *Events* und dessen Verarbeitung, vor allem zeitlich
- *Requests* aggregiert verarbeiten
- Ausnutzung Multi-Threading
- Komplexität
- Scope

12.4 Solution

- Nutze eine Datenstrukture (`PlayMessage`) um Details für einen pendenten Request zu speichern.
- Nutze ein einfacher `array` Ring-Buffer (`Head` und `Tail` anstelle eines `PendingCount`) um die versendeten Messages abzulegen.
  - `Head` und `Tail` jeweils mittels Modulo-Rechnung den array in einer loop (ring) durchlaufen.
- Fange duplizierte Nachrichten bereits beim einreihen von neuen Messages ab.

```
public struct PlayMessage
{
    Guid SoundId;
    int Volume;
}

public class Audio
{
    private const int MAX_PENDING = 16;
    private static readonly PlayMessage[] pending = new PlayMessage[MAX_PENDING];

    private int Head { get; set; }
    private int Tail { get; set; }

    public void PlaySound(Guid soundId, int volume)
    {
```

```
        // Check if any pending requests is the same..,
        for (int i = Head; i != Tail; i = (i + 1) % MAX_PENDING)
        {
            if (pending[i].SoundId == soundId)
            {
                // Use the larger of the two volumes...
                pending[i].Volume = Math.Max(volume, pending[i].Volume);
                // No need to enqueue a duplicated item
                return;
            }
        }

        // Check if Tail would overlap Head
        // TODO: You may wan't tp grow array by N to allow more messages
        Assert((Tail + 1) % MAX_PENDING != Head);

        // Add sound to the end of queue
        pending[Tail].SoundId = soundId;
        pending[Tail].Volume = volume;

        Tail = (Tail + 1) % MAX_PENDING;
    }

    // Called within GameLoop or similar methods
    public void Update()
    {
        if (Head == Tail) return; // No pending updates

        var resource = LoadSound(pending[Head].SoundId);
        var channel = FindOpenChannel();
        if (channel == -1) return;
        StartSound(resource, channel, pending[Head].Volume);

        Head = (Head + 1) % MAX_PENDING;
    }
}
```

12.5 Consequences

12.5.1 Pro

- Der Code für das Empfangen einer Nachricht ist vom Code für das versenden einer Nachricht entkoppelt.
- Wir haben eine Queue für die Koordination zwischen den Sender und Empfänger.
- Die Queue ist entkoppelt vom rest des Programmes.

Somit muss nur noch die “Thread-Safty” im Programm garantiert werden, welcher im Higher-Level gemacht werden kann.

12.5.2 Cons

- Komplexität eher hoch -> Observer- oder Command-Pattern eventuell ausreichend
- Notwendige Information über den aktuellen Context muss immer in der Datenstruktur (`PlayMessage`) übergeben werden, da alles asynchron ist.

12.6 Design Decisions

12.6.1 Inhalt der Queue

12.6.1.1 Events

Ein Event beschreibt ein bereits vergangenes Ereignis wie z.B. “Monster died”. Dieses wird in die Queue eingereiht um von anderen bearbeitet zu werden. Das ist mit einem asynchronen Observer zu vergleichen.

- Mehrere Listeners existieren und es irrellevant ist welcher das Event abbekommt.

- Event-Queues werden oftmals verwendet um Broadcast zu machen. Daher macht man diese global sichtbar.

12.6.1.2 Messages

Eine Message beschreibt eine Aktion, welche in der Zukunft passiert, wie z.B. “play sound”. Man könnte das auch als asynchrone API zu einem Service bezeichnen (Bsp. Command pattern).

- Es ist üblich, dass es hier genau einen Listener gibt, der die Nachricht auf eine spezifische Art und weise verarbeitet.

12.6.2 Empfänger der Queue

12.6.2.1 Single-cast

Single-Cast ist die beste Wahl wenn es darum geht eine Queue in die Klassen API einzupflegen. Der Aufrufer kann dann z.B. eine Methode aufrufen, die dann von einem Empfänger verarbeitet wird.

- Warteschlange nur Implementierungsdetail des Lesers
- Besser gekapselte Queue
- Keine Kommunikation zwischen Empfängern nötig

12.6.2.2 Broadcast

Broadcast ist der empfehlenswerte Event-Queue Lösungsansatz. Wenn ein Event ausgelöst wird können alle Listener die Information verarbeiten.

- Event können bei fehlendem Leser einfach fallen gelassen werden ohne Bearbeitet zu werden.
- Events müssen gefiltert werden, da sonst bei vielen Events mit vielen Lesern massenhaft Event Handler unnötigerweise aufgerufen werden.

12.6.2.3 Work Queue

Auch hier gibt es mehrere Listeners. Der Unterschied ist, dass alle Items in der Queue genau einem Listener zugewiesen werden. Das wird üblicherweise verwendet wenn ein Pool von Anweisungen auf verschiedene Threads verteilt werden müssen.

- Man muss Scheduling betreiben. Das kann mittels Round Robin oder Zufallslogik gemacht werden.

12.6.3 Sender der Queue

12.6.3.1 Ein Sender

Hat Ähnlichkeit zum synchronen Observer Pattern. Ein privilegiertes Objekt generiert Events, welche andere empfangen.

- Herkunft des Items bekannt
- Mehrere Leser sind möglich

12.6.3.2 Mehrere Sender

Als öffentliche Schnittstelle zugänglich ausserhalb der Klasse.

- Man muss auf Zyklen achten
- Referenz auf Sender in Queue-Item hinzufügen

12.6.4 Lebenszyklus der Objekte

Es kann sein, dass Objekte nicht mehr existieren bevor sie in der Queue bearbeitet werden.

12.6.4.1 Pass ownership

Beim Übergeben zur Queue wird die Queue der Besitzer, beim Verarbeiten durch den Empfänger wird dieser der Owner und ist für die Deallokation zuständig.

12.6.4.2 Share ownership

Mit Garbage Collection ist das möglich. Solange das Objekt eine Referenz besitzt lebt es in der Queue.

12.6.4.3 Queue ownership

Die Queue besitzt das Objekt immer und gibt den Empfängern nur Referenzen darauf.

12.7 Aktuelle Praxisbeispiele

- State Pattern besitzt stream von inputs welche mittels Queue verarbeitet werden sollten.
- Go Lang besitzt einen “channel” type welcher genau für message und event queueing existiert.

12.8 Prüfungsfragen

1. Das Event Queue Pattern dient primär zur “statischen” Entkopplung von Sender und Empfänger? *Nein*
2. Versendet eine Queue Nachrichten immer an mehrere Empfänger? *Nein*