# Inhaltsverzeichnis

# 1 POSA

## 1.1 Whole Part Pattern

### 1.1.1 Consequences

#### 1.1.1.1 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

#### 1.1.1.2 Nachteile

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

### 1.1.2 Wichtige Eigenschaften

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

### 1.1.3 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**
3. Komponenten eines Whole-Part Pattens können nicht in anderen Projekten wiederverwendet werden? **Nein**
4. Eine Komplexität des Whole-Part Patterns ist es, Objekte in einzelne Teile zu zerlegen? **Ja**
5. Shared-Parts sind im reinen Whole-Part Pattern erlaubt? **Nein**
6. Das *Whole* muss **nicht** dasselbe Interface wie seine *Parts* implementieren? **Ja**
7. Wodurch unterscheidet sich das Whole-Part Pattern vom Facade Pattern?

- Während beide ein “high-level” Interface für den Zugriff auf eine Aggregation von Softwareelementen definieren ist im Facade Pattern auch der direkte Zugriff auf die einzelnen Elemente erlaubt, im Whole-Part Pattern nur der Indirekte.

2. Nenne 1 jeweils einen Vor und einen Nachteil des Whole-Part Patterns

- Siehe Vor- und Nachteile oben. zB:
  - Vorteil: Austauschbarkeit der Teile
  - Nachteil: Schlechtere Effizienz durch zusätzliche Schicht

1. Ist das Whole-Part Pattern rekursiv anwendbar, z.B. Whole->Whole-Part (Auto->Räder->Felge)? **Ja**
2. Kann man bei der Bottom-Up Variante bereits existierende Objekte immer ohne jegliche Anpassung als Part verwenden? **Nein**

## 1.2 Forwarder-Receiver Pattern

### 1.2.1 Consequences

#### 1.2.1.1 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

#### 1.2.1.2 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)*

### 1.2.2 Wichtige Eigenschaften

- Peers müssen die konkreten Mechanismen zur Kommunikation nicht kennen
- IPC Implementation ist flexibel austauschbar

### 1.2.3 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. Das reine Forwarder-Receiver Pattern setzt voraus, dass ein Peer zur compile-time alle Peers kennt mit denen er kommunizieren muss? **Ja**
4. Ein Peer kann nur ein Server oder Client sein nicht beides? **Nein**
5. Ein Peer **braucht zwingend** je einen *Forwarder* und *Receiver*? Falsch, es kann auch One-Way Kommunikation geben und ein Peer daher nur Receiver oder nur Forwarder haben
6. Das Forward-Receiver Pattern eignet sich nicht für Broadcast? **Falsch** Falsch, je nach Naming-System, z.b. Gruppen oder das mehrere Peers den gleichen Namen haben, ist Broadcast möglich
7. Ein Peer kann jeweils mehrere Forwarder und Reciever haben? **Falsch**
8. Ein Peer welches während der Laufzeit angehängt wird, wird von anderen Peers erkannt und kann von diesen Nachrichten erhalten? **Falsch**
9. Es wird versucht die Namensauflösung, der Verbindungsaufbau und die Deserialisierung vom Peer zu trennen? **Ja**
10. Peer1 kennt die direkte Adresse des Peer2? **Nein**

## 1.3 Blackboard Pattern

### 1.3.1 Consequences

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

#### 1.3.1.1 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.

#### 1.3.1.2 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.

### 1.3.2 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.

### 1.3.3 Prüfungsfragen

1. Knowledge Sources interagieren direkt miteinander um gemeinsam eine Lösung zu finden. **Nein**
2. Das Blackboard Pattern nutzt einen Deterministischen Ansatz. **Nein**
3. Die Lösungssuche ist erst fertig, sobald eine perfekte Lösung gefunden wurde? **Nein**
4. Das Entwickeln eines Blackboard-Systems kann sehr lange dauern? **Ja**
5. *Knowledge Source's* kommunizieren untereinander, um Daten abzugleichen und sich über Änderungen zu informieren Falsch, Knowledge Source's kommunizieren nie direkt miteinander. Sondern lediglich übers via Blackboard in dem sie Datensätze verarbeiten und neue Datensätze hinzufügen
6. Es kann vorkommen, dass nie eine Lösung gefunden wird oder die Lösungssuche sehr lange dauert. Richtig, es ist möglich, dass keine Lösung gefunden wird (auch wenn es eine gibt).
7. Das Blackboard Pattern setzt sich aus einem Blackboard einer Control-Component und einfachen Tasks zusammen? **Falsch** Blackboard, Control Component & Knonwledge Source
8. Alle Knowledge Sourcen können auf die Daten des Blackboard kontrolliert Zugreifen und diese Verändern? **Richtig**
9. Das Pattern ist geeignet, wenn man den Lösungsweg genau kennt? **Nein**
10. Der Controller entscheidet welche Knowledge Source auf das Blackboard schreiben darf. **Ja**

## 1.4 Lazy Acquisition Pattern

### 1.4.1 Consequences

#### 1.4.1.1 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.

#### 1.4.1.2 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.

### 1.4.2 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.

### 1.4.3 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**
3. Lazy Aquisition verhindert lange Wartezeiten bei der Nutzung von Software? **Nein**

4. Ziel ist es teure Ressourcen, so spät wie möglich zu laden? **Ja**
5. Das Lazy Pattern ist in jedem Fall die bessere Variante gegenüber "Beim Start laden" (Eager) **Nein**, es ist ein klassischer Tradeoff. Es muss je nach Situation entschieden werden was besser ist.
6. Die echte Ressource und der Proxy müssen das gleiche Interface für den User bieten **Ja**
7. Die Lazy Aquistion lädt alle Daten früh genug, sodass wenn der User auf diese Zugreifen möchte der Provider nichts mehr tun muss. **Falsch**
8. Das Pattern der Lazy Aquisition besteht aus den vier Klassen: Resource User, Resource Proxy, Resource Provider und Resource? **Richtig**
9. Besitzt das Proxy Objekt ein subset der Funktionen von der echten Ressource? **Nein**
10. Wird beim erstellen und zurückgeben des Proxys auch gleich die Ressource geladen? **Nein**

## 1.5 Coordinator Pattern

### 1.5.1 Consequences

#### 1.5.1.1 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.

#### 1.5.1.2 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.

### 1.5.2 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.

### 1.5.3 Prüfungsfragen

1. Der Client gibt Tasks direkt an einen Participant, welcher diesen dann für ihn atomar ausführt. **Nein**
2. Das Ziel des Coordinator Patterns ist "All or nothing" **Ja**
3. Es ist unmöglich, dass beim two-phase-commit in der commit-Phase ein Fehler auftritt? **Nein**
4. Wenn auf einem Participant ein Task nicht machbar ist, werden die anderen Tasks trotzdem auf allen anderen Participants durchgeführt? **Nein**
5. Das Pattern garantiert, das ein Task in jeden Fall ausgeführt wird **Nein**, es garantiert nur, dass es entweder bei allen oder bei keinem Participant ausgeführt wird.
6. Das Pattern wird oft bei Datenbanken eingesetzt **Ja**
7. In der Prepare Phase werden noch keine permanenten Änderungen an den Teilnehmer vorgenommen **Richtig**
8. Ein Nachteil des Coordinator Patterns ist, dass die Laufzeit bei Erhöhung der Teilnehmer quadratisch zunimmt. **Falsch**
9. Ist die Konsistenz sichergestellt bei einem Fehler in der Commit-Phase beim Three-Phase Commit? **Ja**

10. Werden die Participants vom Coordinator ausgewählt und registiert? **Nein**

## 1.6 Resource Lifecycle Manager

### 1.6.1 Consequences

#### 1.6.1.1 Vorteile

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

#### 1.6.1.2 Nachteile

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

### 1.6.2 Wichtige Eigenschaften

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

## 1.7 Prüfungsfragen

1. Der Resource Lifecycle Manager verwendet einen Resource Provider zum Akquirieren der Ressource. **Richtig**
2. Abhängigkeiten von Resourcen haben einen Einfluss auf ihren Lifecycle. **Richtig**
3. Die Idee hinter dem Resource Lifecycle Manager Pattern ist es, die Verwendung und Verwaltung von Ressourcen zu trennen? **Ja**
4. Das Resource Lifecycle Manager Pattern eignet sich besonders gut für «large-scale» Systeme? **Ja**
5. Es darf in einem System mehrere RLMs geben **Ja** (siehe Solution)
6. Das Resource Lifecycle Manager Pattern verhindert Single Point of Failures **Nein**, im Gegenteil (siehe Nachteile)
7. Die Ressourcen einer Applikation können nur von einem Resource Lifecycle Manager verwaltet werden. **Falsch**
8. Durch den Resource Lifecycle Manager können Ressourcen wieder freigegeben werden sobald sie nicht mehr benötigt werden. **Richtig**
9. Kann der Resource Lifecycle Manager Optimierungen wie Caching / Lazy-Loading implementieren? **Ja**
10. Wird eine resource immer an den resource user zurückgegeben? **Nein** Wenn es zu wenig hat nicht.

# 2 MAP

## 2.1 Microservice API Patterns

### 2.1.1 Atomic Parameter

#### 2.1.1.1 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.

#### 2.1.1.2 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.

### 2.1.2 Parameter Tree

#### 2.1.2.1 Consequences

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

##### 2.1.2.1.1 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.

##### 2.1.2.1.2 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.

### 2.1.3 Parameter Forest

#### 2.1.3.1 Consequences (Parameter Forest)

##### 2.1.3.1.1 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.

##### 2.1.3.1.2 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.

### 2.1.4 Pagination

#### 2.1.4.1 Consequences (Pagination)

##### 2.1.4.1.1 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.

##### 2.1.4.1.2 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.

#### 2.1.5 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.
3. Ein Parameter Tree Pattern, das nur aus Atomic-Parametern besteht ist äquivalent zum Atomic Parameter List Pattern? **Ja**
4. Wird Pagination vor allem dazu verwendet, Datensets besser zu strukturieren? **Nein**
5. Das Verwenden von primitiven Datentypen bei "Atomic Parameter" erhöht die Kopplung. _-> **Falsch**. Die Verwendung von primitiven Datentypen verringert die Kopplung.
6. Die "Atomic Parameter List" setzt sich aus mehreren Single Scalar Representations zusammen. _-> **Richtig**.
7. Das unendliche Scroll-Feature von Sozialen Netzwerken ist ein typisches Beispiel für die Anwendung des Pagination Patterns. _-> **Richtig**.
8. Bei einem Parameter Tree werden mehrere Root Elemente definiert, während bei einem Parameter Forrest nur ein Root Element definiert wird. _-> **Falsch**.
9. Ist es eine gute Idee beim Atomic Parameter List-Pattern ein Array als Parameter zu verwenden? **Nein**
10. Im Atomic Parameter Pattern geht es grundsätzlich darum, dass die Parameter atomar verarbeitet werden können? **Nein**
11. Wurden bei der folgender Nachrichtendarstellung alle Vorgaben des Parameter Tree Pattern eingehalten?

```
{
    "claim": {
        "id": "0afeb849-6d63-40b6-b52f-21dee16fdda5",
        "evidence":[{
                "name":"example1"
            }
        ]
    },
    "link":"https://localhost/0afeb849-6d63-40b6-b52f-21dee16fdda5"
}
```

**Nein**, da in der Datenstruktur mehr als ein Root-Element vorhanden ist. 4. Der Parameter Forest Pattern ist die ausgeprägteste Form der vorgestellten Patterns (dieses und letztes Mal) und daher das ratsamste Pattern. Die Anwendung ist unbedenklich und stellt keine Probleme dar.
**Nein**, auch dieses Pattern hat seine Problemchen. Nur weil das Pattern die meisten Möglichkeiten bietet, muss man es nicht exzessiv und unbedacht für jeden Fall anwenden. Das Hauptproblem des Patterns besteht darin, dass eher mehr Daten übertragen werden, als wirklich benötigt werden. Tief verschachtelte Datenstrukturen werden schnell komplex und stehen der Verständlichkeit im Weg.

1. What does an API-Provider define and provide Instead of a complex API? Solution: a List with Key-Value pair in which the key is a single (or multiple) scalar(s)
2. Does the Atomic Parameter Pattern provide Developer convenience and experience? Solution: **yes**, except for security
3. Does the pagination pattern decrease overall performance? -> **No**
4. Does the pagination pattern increase security? -> **Yes**
5. Bilden Interface-Responsibility (Schnittstellenverantwortung), Structure (Struktur) und Delivery-Quality (Lieferqualität) die drei Kernkategorien der sprachlichen Organisation? **Ja**
6. Können mit dem Atomic Parameter (List) Daten strukturiert werden? **Nein**. Für strukturierte Daten müssen die Pattern Parameter Tree oder Parameter Forest verwendet werden.

7. Kann ein Tupel oder eine Liste von Zahlen als Parameter-Forest aufgefasst werden? **Nein**. Zahlen sind atomatmoare Typen. Aus ihnen können aber Parameter-Tree oder Parameter-Forest zusammengesetzt werden.
8. Sind verschachtelte Objekte eine Sonderform von Parameter-Collections? **Nein**. Parameter-Collections sind eine Sonderform von verschachtelten Objekten.

## 2.2 Interface Evolution Patterns

### 2.2.1 Consequences

#### 2.2.1.1 Pros

- Gute Planbarkeit aufgrund der bereits bekannten Supportdauer

#### 2.2.1.2 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

### 2.2.2 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**
3. Eine beliebige API Versionierung erlaubt es dem Client die Kompatibilität zu erkennen? _-> **Falsch**.
4. Beim Pattern "Two in Production" weiss der Client von Anfang an, bis wann er auf eine neuere Version der API upgraden muss? _-> **Richtig**.
5. Dürfen für eine Anfrage mehrere Version-Identifiers verwendet werden? Z.B. ein GET v1/customers/1234 mit Accept: text/json; version=3.1 **Ja**. Die Implementation kann sich unabhängig von der Schnittstelle weiterentwickeln. Was zu zu mehreren Versions-Identifiers führt.
6. Ein API Provider darf nach Limited Lifetime Guarantee Pattern keinerlei Änderungen in der API vornehmen. **Nein**. Die API darf vom Provider verändert werden, insofern sie Rückwärtskompatibel für bestehende Clients bleibt. Sobald das Ablaufdatum der API eintritt, steht es dem Provider frei, den Support der API einzustellen.
7. Can this Pattern also be used outside of APIs? => **Yes**
8. Wenn ich meiner Software eine zusätzliche Funktion hinzufüge, welche andere Funktionen nicht beeinflusst, ändere ich die Version wie folgt x.y.z -> x.(y+1).z? **Ja**. Die Anpassung der Nebenversion ist in diesem Fall notwendig.
9. Wenn ein API-Anbieter seine Clients über die Laufzeit einer angebotenen Version informieren möchte, genügt es, wenn er für einen fixen Zeitraum keine Unverträglichkeiten bei der publizierten Schnittstelle verspricht (Garantie durch Anbieter). **Nein**. Die Garantie alleine reicht nicht. Alle publizierten Schnittstellen müssen mit einem Ablaufdatum ergänzt werden.

## 2.3 Game Loop

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

#### 2.3.1.1 Benefits

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

#### 2.3.1.2 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

### 2.3.2 Take a little nap (Fixed time step with synchronization)

#### 2.3.2.1 Benefits

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

#### 2.3.2.2 Problems

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

### 2.3.3 One small step, one giant step (Variable time step)

#### 2.3.3.1 Benefits

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

#### 2.3.3.2 Problems

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

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

#### 2.3.4.1 Benefits

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

#### 2.3.4.2 Problems

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

### 2.3.5 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**
3. Ein Game Loop mit einem Synchronisierungsschritt am Ende kann nicht verhindern, dass ein Spiel sich auf alter Hardware verlangsamt. -> Korrekt, wenn der update Schritt länger geht als die Framelänge hilft die Synchronisierung nichts.
4. Ein klassischer Game Loop wartet auf User Input bevor der Loop weiter ausgeführt wird. -> **Falsch**. Wenn auf User Inputs gewartet werden würde, würde das Bild einfrieren und der Game State würde sich nicht weiter anpassen bis der nächste Input getätigt würde.
5. Beim Game Loop können nur Probleme entstehen, wenn die Hardware zu langsam ist? **Falsch**, auch wenn die Hardware zu schnell ist, kann es zu Problemen führen.
6. Es gibt 2 wichtige Bestandsteile beim Game Loop. Nicht auf Benutzer Eingaben warten und das Spiel muss mit einer konstanten Geschwindigkeit laufen **Richtig**, siehe Key Parts
7. A variable time step is a fine solution if you are planning to run your app on multiple different devices **False**
8. It is recommended to always take the platform event loop **False** (It depends)
9. Das Game-Loop-Pattern schreibt vor wie Verzögerungen behandelt werden? **Nein**, das Game-Loop-Pattern bietet mehrere Subpatterns, welche verschiedene Varianten aufzeigen.

10. Das Subpattern «Play catch up» berücksichtigt bei unterschiedlichen Geräten einen gleichen Spielverlauf? **Ja**, sowie das «Stuck in the middle» Subpattern.

## 2.4 Component Game Pattern

## 2.5 Consequences

### 2.5.1 Pros

- Bessere Wiederverwendbarkeit von Logik der einzelnen *Domains*
- Reduziert Kopplung der einzelnen *Domains* massiv
- Austauschbarkeit der Komponenten

### 2.5.2 Cons

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

### 2.5.3 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**
3. Die erschwerte Kommunikation zwischen Komponenten im Komponentenpattern ist eine negative Konsequenz. -> **Richtig**
4. Die Komponenten im Komponentenpattern können weniger einfach wiederverwendet werden als bei klassischen Vererbunghierarchien -> **Falsch**
5. Erhöht das Component Pattern die Kopplung zwischen verschiedenen Klassen? **Nein**, es sorgt für eine lose Kopplung.
6. Macht es in jedem Fall Sinn das Component Pattern anzuwenden? **Nein**, nicht in jedem Fall. Bei einer grossen Codebasis macht es oft durchaus Sinn. In kleinen Projekten kann es dazu kommen, dass das Pattern ein Problem löst, welches gar nicht bestanden hat.
7. Goal of this Pattern is High Coupling and Low Cohesion? => **False**
8. The Component Pattern is only useful when programming a game. => **False**
9. Das Component Pattern löst das Problem von zu gekoppelten Klassen? **Ja**. Durch die Anwendung des Component Patterns können Klassen nach Domänen geteilt werden.
10. Die Basisklasse, welche alle Komponenten zusammenbindet, kennt die genauen Implementationen der Komponenten und kann dadurch einfach in das Verhalten der Komponenten eingreifen? **Nein**. Die Basisklasse kennt die genauen Implementationen nicht und sollte in das Verhalten der Komponenten nicht eingreifen. Das Prinzip des Component Pattern ist die Trennung nach Domänen.

## 2.6 Event Queue

## 2.7 Consequences

### 2.7.1 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.

### 2.7.2 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.

### 2.7.3 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**
3. In einer einfachen Message Queue kann "gedrängelt" werden. -> **Falsch**. Es gibt in einfachen Message Queues keine Priorisierungen.
4. Message Queues werden oft mit Circular Arrays (a.k.a Ring Buffers) implementiert. -> **Richtig**.
5. Globale Queues bergen Performanceprobleme? **Nein**, nicht per se. Wenn man es richtig macht, schon. Soll die Event-Queue threadsafe sein, geht aufgrund von synchronisationsmechanismen Performance verloren. Dies kommt aber nicht gratis daher, das muss bewusst gemacht werden. Ist eine Queue nicht threadsafe, so birgt sie andere Gefahren, schmälert aber nicht die Performance an sich. Hier eignet sich eventuell der Einsatz mehrerer Event-/Message-Queues.
6. Dieses Pattern entkoppelt zeitliche Ausführung und die Komponenten voneinander **Ja**, bei diesem Pattern handelt es sich und die Entkoppelung der Komponenten an sich und die Ausführung ist ebenso zeitlich voneinander entkoppelt. Gerade bei Events ist nicht garantiert, dass diese auch direkt abgearbeitet werden.
7. Events use interrupts **No**
8. If I only want to decouple Elements which receive messages from its sender, I could also use an Observer- or Command-Pattern. **Yes**
9. Nicht jede Event-Queue braucht eine listenartige Struktur, in welcher die Events gehalten werden? **Nein**. Jede Event-Queue benötigt irgend eine Art Liste, in welcher die Events abgespeichert werden.
10. Klassische synchrone Events (auf Systemen ohne Event-Queue) sind datenlastiger als Events in einer Event-Queue (auf Systemen mit Event-Queue)? **Nein**. Events in einer Event-Queue (oder auf einem System mit Event-Queue) sind prinzipiell datenlastiger. Auf Systemen ohne Event-Queue reicht eine Art Eventtyp aus, da in den Umsystemen direkt nachgeschaut werden kann, was sich geändert hat. Auf Systemen mit Event-Queue können sich die Umsysteme bereits verändert haben, oder existieren bereits nicht mehr, diese Daten müssen zusätzlich im Event gespeichert werden.