

Advanced Software Engineering

Projektarbeitsdokumentation

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges

Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Dominik Ochs

Abgabedatum:

XX. Mai 2023

Bearbeitungszeitraum:

01.10.2022 - XX.05.2023

Matrikelnummer, Kurs:

2847475, TINF20B2

Gutachter der Dualen Hochschule:

Dr. Lars Briem

Inhaltsverzeichnis

Abbildungsverzeichnis	II
Quellcodeverzeichnis	III
1. Einführung	1
1.1. Übersicht über die Applikation	1
1.2. Wie startet man die Applikation?	2
1.3. Wie testet man die Applikation?	3
2. Clean Architecture	4
2.1. Was ist Clean Architecture?	4
2.2. Analyse der Dependency Rule	4
2.3. Analyse der Schichten	6
3. SOLID	10
3.1. Analyse Single-Responsibility-Principle (SRP)	10
3.2. Analyse Open-Closed-Principle (OCP)	12
3.3. Analyse Interface-Segregation-Principle (ISP)	15
4. Weitere Prinzipien	19
4.1. Analyse GRASP: Geringe Kopplung	19
4.2. Analyse GRASP: Hohe Kohäsion	23
4.3. Don't Repeat Yourself (DRY)	24
5. Unit Tests	28
6. Domain Driven Design	29
7. Refactoring	30
7.1. Zusammenfassung	30
7.2. Kritische Reflexion	30
7.3. Zukunftsaussicht	30
8. Entwurfsmuster	31
A. Anhang	IV
A.1. Architekturen	IV
A.2. Quelltext-Implementation	V

Abbildungsverzeichnis

1.1. Spielphasen.	2
2.1. UML-Diagramm 1 zum Einhalten der Dependency Rule von <i>Camp</i>	5
2.2. UML-Diagramm 2 zum Einhalten der Dependency Rule von <i>Game</i>	6
2.3. Beispielklasse der Plugin-Schicht: <i>ErrorBuilder</i>	7
2.4. Beispielklasse der Domain-Schicht: <i>ResourceStash</i>	9
3.1. UML-Diagramm von <i>ase.plugin.cli.parsers.DiceParser</i>	10
3.2. UML-Diagramm von <i>application.RollHandler</i>	11
3.3. Abbildung 3.2 aufgeteilt in <i>EndeavorHandler</i> und <i>EncounterHandler</i> . . .	12
3.4. UML-Klassendiagramm vom <i>RollDxCommand</i> und weiteren Commands, wovon die meisten allerdings ausgelassen wurden der Übersichtlichkeit wegen.	13
3.5. UML-Klassendiagramm von <i>CardInvalidator</i> und <i>Card</i>	14
3.6. Abbildung 3.5 mit Card-Interface und Implementationen statt Card-Enum und CardCategory. Aus Übersichtlichkeit wurden nicht alle Implementationen eingezeichnet.	15
3.7. UML-Diagramm von <i>Buildable</i> , <i>Endeavor</i> , <i>Tool</i> und <i>Building</i> und deren Implementationen.	16
3.8. UML-Diagramm von <i>Deck</i> und dessen Implementationen.	16
3.9. Abbildung 3.8 mit abgespaltetem <i>Serializable</i> -Interface.	18
4.1. UML-Diagramm von <i>ase.application.PersistenceWriter</i>	20
4.2. UML-Diagramm von <i>ase.domain.crafting.Workbench</i>	22
4.3. Abbildung 4.2 mit eingezogenem <i>ResourceConsumer</i> -Interface zur Auflösung der Kopplung.	23
4.4. UML-Diagramm von <i>ase.domain.crafting.ResourceRequirement</i>	24

Quellcodeverzeichnis

4.1	DRY-Code der <i>Game</i> -Klasse zum Commit b4399813.	25
4.2	DRY-Code der <i>Game</i> -Klasse zum Commit e246d77b.	26

1. Einführung

1.1. Übersicht über die Applikation

Bei dieser Applikation handelt es sich um ein singleplayer Kartenspiel. Hier versucht ein Spieler auf einer einsamen Insel zu überleben und von ihr zu flüchten, indem er Karten zieht und Ressourcen (*wood, plastic, metal*) sammelt, um Gegenstände zu bauen (*Scavage*), die ihn vor seinen Feinden (*Encounter*) schützen oder ihm eine Rettung (*Endeavor*) von der Insel ermöglichen. Während des Spiels trifft der Spieler auf Tiere (*spider, tiger, snake*), die ihm gesammelte Ressourcen kosten können, oder Katastrophen, die Ressourcen zerstören. Das Spiel endet, wenn der Spieler sich retten konnte oder keine Aktionen mehr möglich sind.

Das Kartenspiel hat verschiedene Spielphasen (s. Abbildung 1.1), in denen der Spieler unter bestimmten Bedingungen Karten ziehen und Gegenstände bauen kann (*Scavage*). Wenn ein Gegenstand aus der Kategorie Rettungen (*Endeavor*) gebaut wurde, muss der Spieler würfeln, um zu entscheiden, ob die Rettung erfolgreich ist. Ein Segelboot und ein Hanggleiter stellen eine Rettung dar, wenn beim Würfeln eine bestimmte Augenzahl erzielt wird. Wenn der Rettungsversuch fehlschlägt, kann der Spieler weiter Karten ziehen und Gegenstände bauen, solange Karten im Stapel vorhanden sind. Das Bauen eines Dampfschiffs und eines Heißluftballons ist nur mit einer Feuerstelle möglich und garantiert eine erfolgreiche Rettung. Wenn ein Tier gezogen wird, muss der Spieler gegen es kämpfen (*Encounter*) und auch hier bestimmt Würfeln über Sieg oder Niederlage. Für unterschiedliche Aktionen werden unterschiedliche Würfel (*vier-, sechs-, acht-seitig*) verwendet. Das Spiel endet, wenn keine weiteren Aktionen mehr möglich sind oder wenn eine garantierte Rettung erfolgt ist. Wenn das Spiel endet (*End*), bleibt es in diesem Zustand, bis ein neues Spiel gestartet wird, bis das Spiel neu initialisiert wird oder die Anwendung beendet wird.

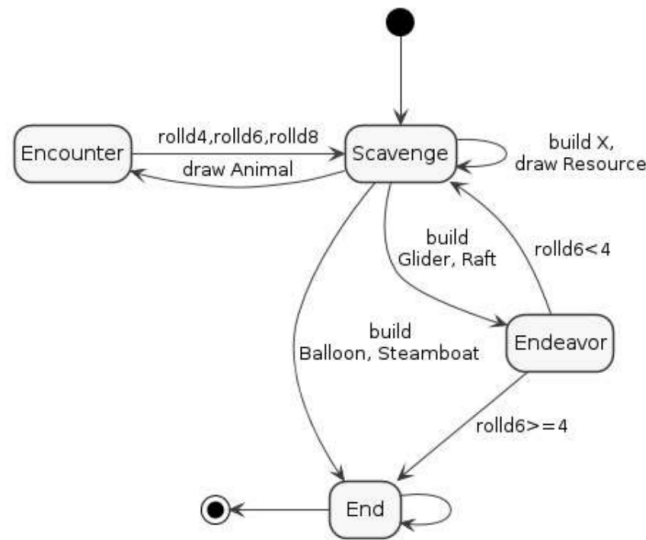


Abbildung 1.1.: Spielphasen.

1.2. Wie startet man die Applikation?

Vorraussetzung ist **Java 17**¹.

Wahlweise wird eine gängige Java-IDE² oder eine *maven*-Installation benötigt.

Über Maven:

Im Wurzelverzeichnis folgende Maven-Befehle ausführen:

```
mvn compile
mvn package
java -jar target/ase-1.0-SNAPSHOT.jar
```

¹Check mit `java -version`.

²Hier wurde *Jetbrains' IntelliJ* verwendet.

Über IDE:

Projekt mit IDE im Wurzelverzeichnis öffnen. Zu folgendem Pfad navigieren

```
./src/main/java/de/dhbw/karlsruhe/ase/plugin/cli
```

und dort die *main*-Methode der Klasse *Main* ausführen lassen.

Erste Schritte:

Nun läuft das Command Line Interface der Applikation und es kann mit

```
help
```

eine Übersicht über die möglichen Befehle (Name, Regex, Beschreibung) aufgerufen werden, oder mit

```
start?
```

ein Spiel mit zufälligem Kartenstapel gestartet werden. Daraufhin kann mit

```
draw
```

begonnen werden Karten zu ziehen.

1.3. Wie testet man die Applikation?

Über Maven:

Im Wurzelverzeichnis:

```
mvn test
```

Über IDE:

Die Tests befinden sich unter

```
./src/test/java
```

und können dort mit einer IDE-Funktion ausgeführt werden.

Es existieren Blackbox-Integration-Tests und Unit-Tests, die aber keine besonderen Anforderungen außer eine **JUnit5**-Abhängigkeit haben.

2. Clean Architecture

2.1. Was ist Clean Architecture?

Clean Architecture ist ein allgemeines Konzept zum Design von Softwarearchitekturen, das darauf abzielt, die Struktur einer Anwendung so zu gestalten, dass sie unabhängig von ihren Benutzerschnittstellen, Datenquellen und anderen äußeren Faktoren bleibt. Dies bedeutet, dass die verschiedenen Komponenten und Funktionen einer Anwendung in abgekapselten Schichten angeordnet werden (*Onion-Architektur*), um sicherzustellen, dass Veränderungen in einer Schicht die tieferliegenden Schichten nicht beeinflussen. Äußere Schichten haben dabei Abhängigkeiten von tieferen Schichten aber niemals umgekehrt. Aufrufe von tieferen Schichten an äußere Schichten werden über Abstraktionen (*Dependency Inversion* und *Injection*) realisiert. Tiefere Schichten sind dabei langlebiger als Schichten weiter außen.

Dies führt zu einer Anwendung, die flexibler und leichter zu entwickeln und zu warten ist, da Änderungen an einem Teil der Anwendung die tieferliegenden Schichten nicht berühren.

2.2. Analyse der Dependency Rule

Schichten im Sinne der Dependency Rule sind hier `plugin`, `application`, `domain` und `abstraction` aufsteigend geordnet nach zunehmender Tiefe in der Onion-Architektur¹. Sie werden durch gleichnamige Packages, die parallel unter `de.dhbw.karlsruhe.ase` zu finden sind, erschöpfend repräsentiert. Es sind Abhängigkeiten von äußerden Schichten in tiefere Schichten erlaubt, aber nicht umgekehrt. Diese Regel wird für die oben genannten Schichten immer eingehalten, was die folgenden beiden Beispiele illustrieren.

¹Siehe ?? für mehr Informationen.

Positiv-Beispiel 1:

Abbildung 2.1 zeigt das verlangte UML-Diagramm der Beispielklasse 1 *Camp* der Schicht *domain*, welches die Dependency Rule einhält. Wie im Diagramm zu sehen ist hängt *Camp* ausschließlich von Klassen in der selben Schicht (*domain*) ab. *Crafting* ist dabei keine Schicht sondern ein *Aggregat*². Schichten sind ausschließlich die oben genannten. Weiter zu sehen ist, dass ausschließlich Klassen in *application* von *Camp* abhängen, aber keine aus höheren Schichten (was in diesem konkreten Fall nur *abstraction* sein könnte).

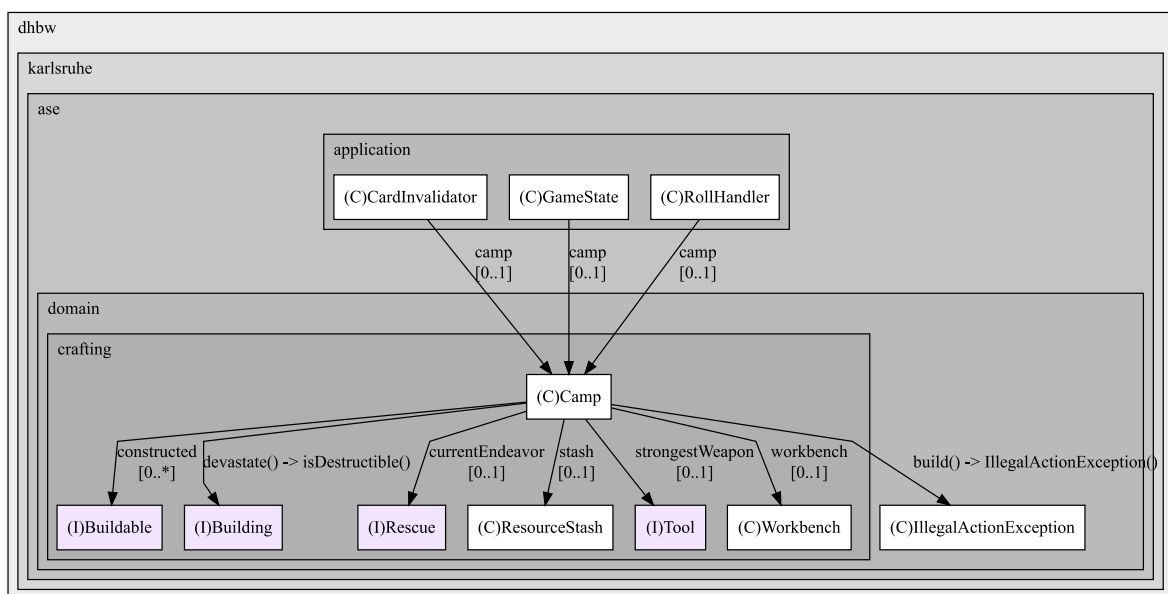


Abbildung 2.1.: UML-Diagramm 1 zum Einhalten der Dependency Rule von *Camp*.

Positiv-Beispiel 2:

Abbildung 2.2 zeigt ein weiteres positives Beispiel für die Dependency Rule, da diese in dieser Softwarearchitektur nicht verletzt wird. Abgebildet ist die Klasse *Game*, welche lediglich von Klassen aus der eigenen Schicht (*application*) und Klassen aus der tieferen Schicht *domain* abhängt. Abhängig von *Game* sind nur Klassen aus der Schicht *plugin*, bzw. dem konkreten *cli*-Plugin.

²Siehe Kapitel 6.

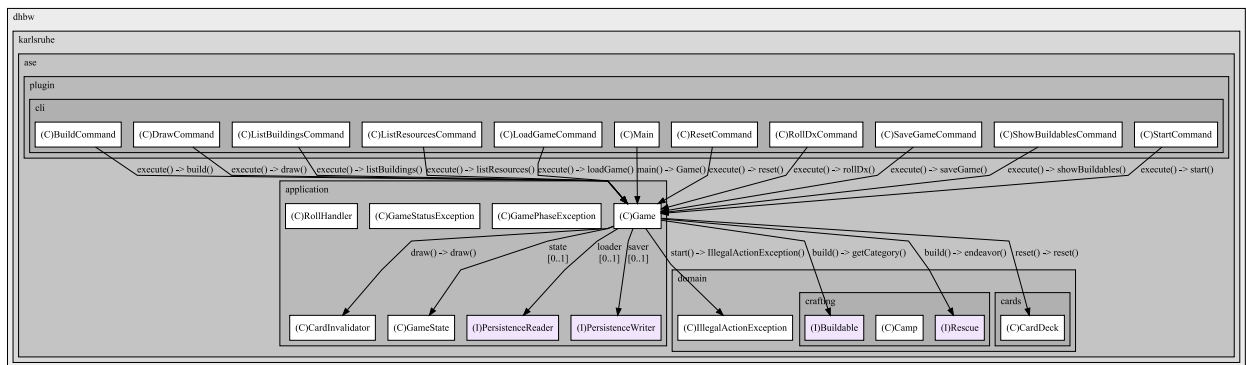


Abbildung 2.2.: UML-Diagramm 2 zum Einhalten der Dependency Rule von *Game*.

2.3. Analyse der Schichten

In dieser Softwarearchitektur gibt es folgende Schichten aufsteigend sortiert nach Tiefe: *plugin*, *application*, *domain* und *abstraction*. Hierbei stellt *plugin* die äußerste und *abstraction* die tiefste Schicht dar.

Schicht Plugin:

Bei einem Refactoring der Plugin-Schicht ist aufgefallen, dass viele der (Fehler-)Hinweise, die aufgrund von falschem Befehlssyntax oder einem falschen Befehl an den User über das CLI ausgegeben werden, keine gemeinsame einheitliche Form aufwiesen, da diese direkt als String ausgegeben wurden. Um eine einheitliche Fehlermeldung zu gewährleisten wurde der *ErrorBuilder* eingefügt, dessen Aufgabe es ist, die Fehler einheitlich zu formatieren und auszugeben. Abbildung 2.3 zeigt das UML-Klassendiagramm dieser Klasse. Sie wird im gesamten CLI-Plugincode verwendet um (Fehler-)Hinweise auszugeben.

Über verschiedene Konstruktoren kann ein Grund für den (Fehler-)Hinweis und eine mögliche Abhilfeempfehlung eingegeben werden. Mit der *print*-Methode kann der erzeugte (Fehler-)Hinweis dann formatiert an den User ausgegeben werden. Hierzu wird aus Gründen der Testbarkeit an die Funktion *printError* des Proxys *Terminal* delegiert.

Diese Klasse befindet sich im Plugin-Layer (insb. im CLI-Plugin), da die konkrete Ausgabe an den User in Form von Text von dem CLI abhängt. Die high-level Fehlermeldungen die von den tieferen Schichten über Exceptions realisiert sind können von jedem Plugin

anders verarbeitet und an den User weitergegeben werden. In der CLI funktioniert dies mit dem `ErrorBuilder`, während es mit einem denkbaren GUI-Plugin über bspw. ein Popup funktionieren könnte. Der `ErrorBuilder` ist lediglich für die Ausgabe an den Nutzer zuständig und enthält keine Fehler-Logik und gehört somit in die Plugin-Schicht.

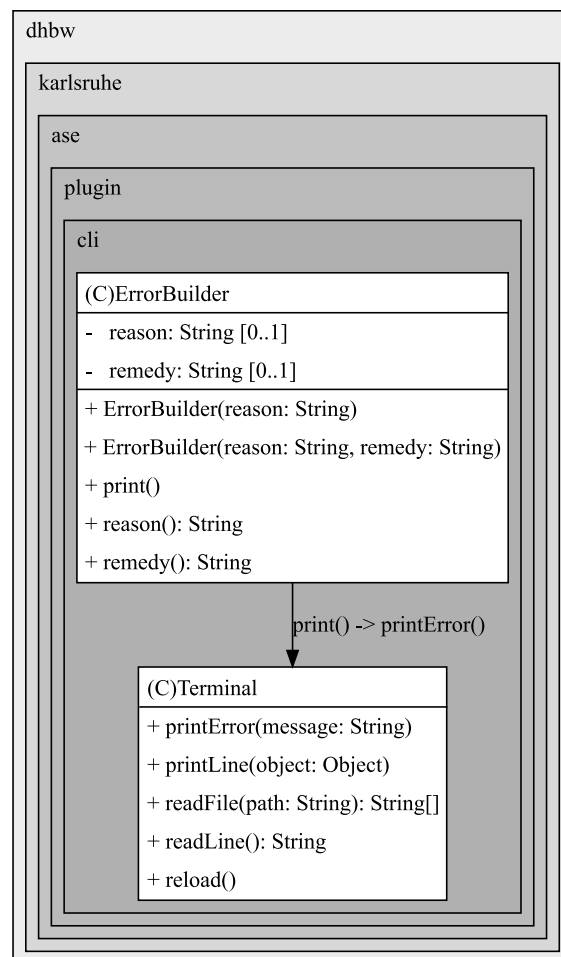


Abbildung 2.3.: Beispielklasse der Plugin-Schicht: `ErrorBuilder`.

Schicht Domain:

Der *ResourceStash* ist fester Teil der Domain-Schicht und stellt hier ein Wrapper mit Methodennamen in Domänensprache für eine Resource-Deque dar. Abbildung 2.4 zeigt das zugehörte UML-Klassendiagramm. Der `ResourceStash` beinhaltet und verwaltet alle Ressourcen, die der Spieler im Verlauf des Spiels ansammelt. Durch eine Katastrophe

oder das Verlieren gegen ein Tier wird der ResourceStash zerstört (*devastate*) und alle ungeschützten Ressourcen gelöscht. Das Erbauen eines *Shacks*³ erlaubt es die obersten $n \in [0; \text{inf})$ Ressourcen zu schützen (*protectTopMostNResources*), sodass diese nach einem *devastate* übrig bleiben. Zusätzlich können Ressourcen hinzugefügt, konsumiert oder deren Vorhandensein überprüft werden. *Camp* und *Workbench* teilen sich eine Referenz auf denselben Stash.

Die Klasse ist als Teil des Kartenspiels (der Domäne) im Code natürlich in der Domain-Schicht angesiedelt und wird von anderen Domänenklassen genutzt. Es gibt keine Möglichkeit die Klasse in einer der anderen Schichten sinnvoll anzusiedeln.

³nicht in der Abb.

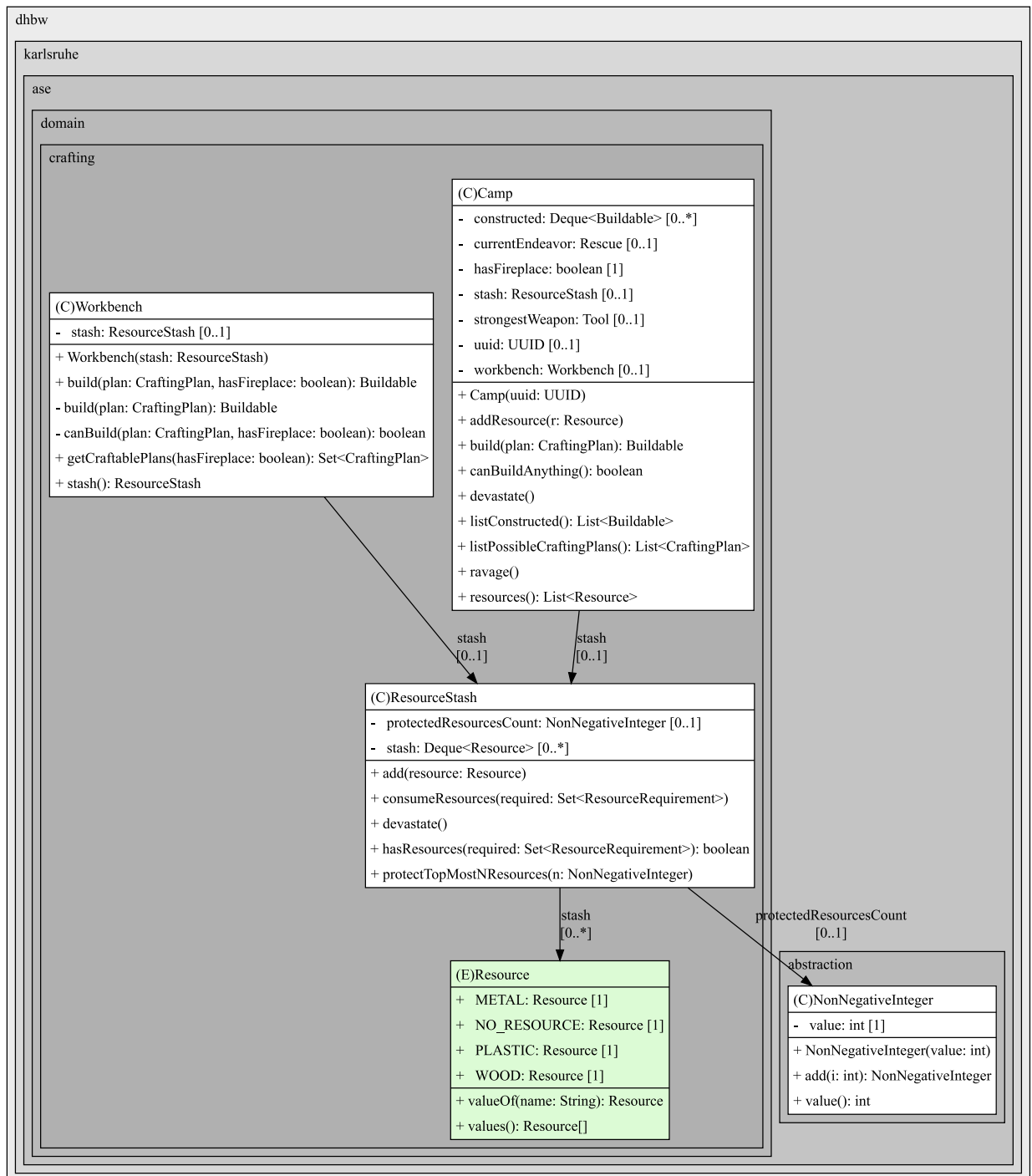


Abbildung 2.4.: Beispielklasse der Domain-Schicht: ResourceStash.

3. SOLID

3.1. Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel:

Abbildung 3.1 zeigt das UML-Klassendiagramm des *DiceParser*, welcher in der Plugin-Schicht unter `plugin.cli.parsers` zu finden ist und das Positiv-Beispiel darstellt. Dieser implementiert das *Parser*-Interface aus `plugin.cli`. Die einzige Aufgabe des *DiceParsers* ist es die Usereingabe zum Erstellen eines Würfelwurfs aus Text zu parsen und das entsprechende Objekt zu kreieren. Hierzu wird der *Regex*-Matcher, der bereits den Syntax überprüft hat an die *parse*-Methode übergeben und daraus die Argumente extrahiert, um den *Roll* zu erstellen.

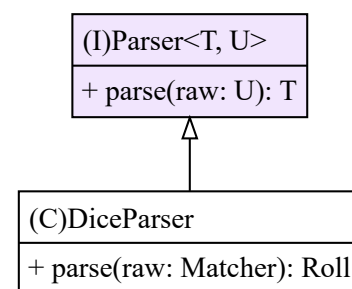
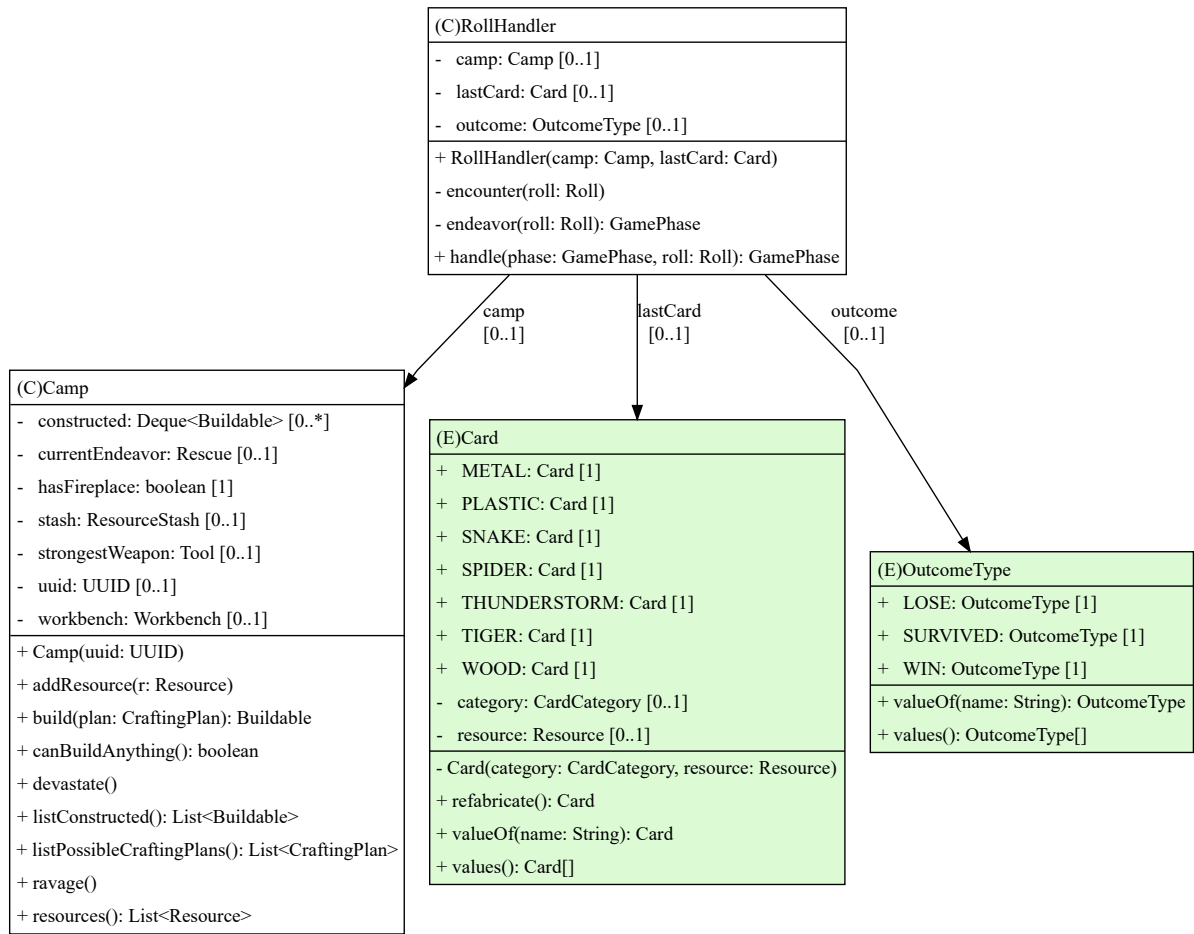


Abbildung 3.1.:
UML-Diagramm von
ase.plugin.cli.parsers.DiceParser.

Negativ-Beispiel:

Abbildung 3.2 zeigt das UML-Klassendiagramm des *RollHandler*, welcher in der Applikations-Schicht die Würfelwürfe für die Rettungen (*Endeavor*) und Kämpfe (*Encounter*) bearbeitet. Hierbei entscheidet die öffentliche *handle*-Methode, ob es sich um einen *Encounter* oder ein *Endeavor* handelt (dies ist allerdings eigentlich schon bei Aufruf dieser bekannt) und führt die entsprechende private Methode aus. Die Klasse hat also zwei Aufgaben (Responsibilities).

Um dies zu lösen kann einfach die Klasse *RollHandler* in zwei Klassen aufgeteilt werden - den *EncounterHandler* und *EndeavorHandler* - die nun jeweils nur noch genau eine Aufgabe haben und somit das SRP einhalten, wie Abbildung 3.3 zeigt.

Abbildung 3.2.: UML-Diagramm von *application.RollHandler*.

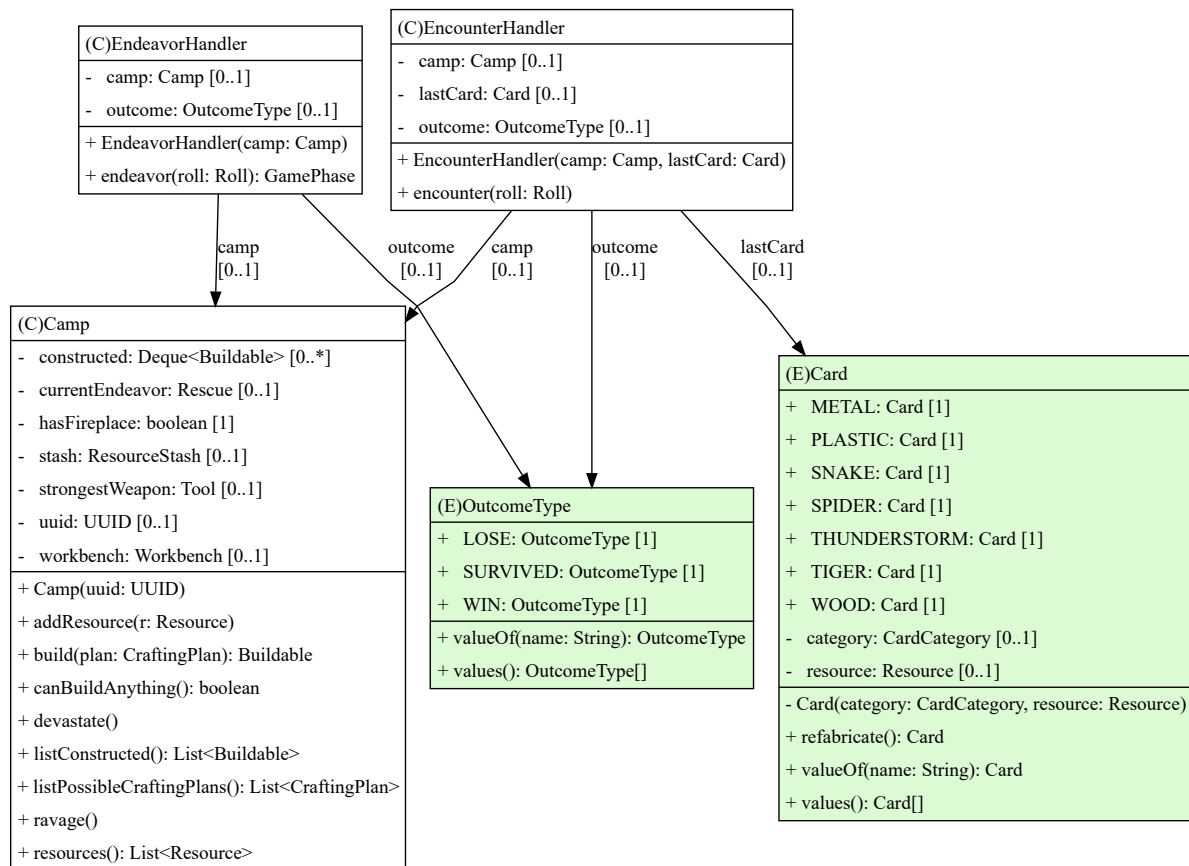


Abbildung 3.3.: Abbildung 3.2 aufgeteilt in *EndeavorHandler* und *EncounterHandler*.

3.2. Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel:

Das Positiv-Beispiel zum OCP wird konstituiert durch das *Command*-Interface aus `ase.plugin.cli` und dessen Implementationen (z.B. *RollDxCommand*) aus `ase.plugin.cli.com` wie gezeigt in Abbildung 3.4. Die Main-Klasse kennt nur das *Command*-Interface und ruft darauf die *execute*-Methode auf, die dann in den unterschiedlichen *Command*-Implementationen verschiedene Wirkungen auf das übergebene *Game* haben. Dies ist auch die Begründung, wieso das OCP hier erfüllt wird: Um einen neuen Befehl zu implementieren muss lediglich eine weitere Klasse hinzugefügt werden, die das *Command*-Interface

implementiert. Es muss dazu keine der bestehenden Command-Klassen angepasst oder geändert werden.

Das OCP ist hier sehr sinnvoll, da für neue Features sehr wahrscheinlich regelmäßig neue Commands hinzugefügt werden müssen und dies soll somit möglichst einfach umsetzbar sein und keinen bestehenden Code breaken. Zuvor wurden die unterschiedlichen Befehle über zahlreiche `switch`-Blöcke realisiert (siehe z.B. Commit 034a5c28), was Änderungen des Programmcodes an vielen Stellen nötig machte, um einen neuen Befehl hinzuzufügen. Außerdem wurden Runtime-Exceptions geworfen, wenn vergessen wurde den Code an einer Stelle anzupassen. Durch das neue System müssen keine Änderungen (geschlossen für Änderungen) an vielen Stellen mehr durchgeführt werden, sondern nur noch Additions durchgeführt werden (offen für Erweiterung).

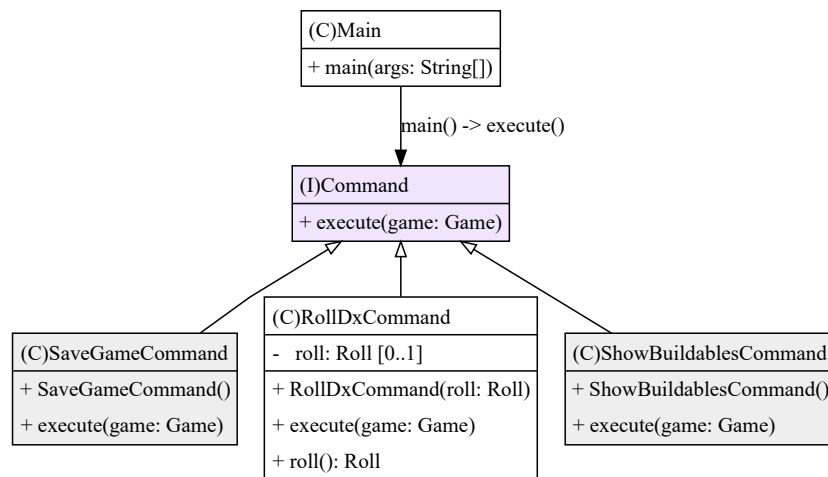


Abbildung 3.4.: UML-Klassendiagramm vom *RollDxCommand* und weiteren Commands, wovon die meisten allerdings ausgelassen wurden der Übersichtlichkeit wegen.

Negativ-Beispiel:

Abbildung 3.5 zeigt das Negativ-Beispiel für das OCP. Die hier relevanten Klassen sind der *CardInvalidator* und das *Card*-Enum, welches eine *CardCategory* und eine *Resource* enthält. Der *CardInvalidator* zieht bisher eine Karte vom *CardDeck* mit der *draw*-Methode und invalidiert die Karte, indem er deren Effekt einlöst. Dies wird mithilfe eines `switch`-Statements über die *CardCategory* der Karte entschieden. Wenn nun also eine neue Karte mit einem neuen Karteneffekt hinzugefügt werden soll, muss also die *draw*-Methode

verändert werden. Das OCP ist also nicht gewahrt.

Abbildung 3.6 zeigt eine mögliche Lösung, um das OCP einzuhalten: Das Card-Enum wurde mit einem Card-Interface mit der *invalidate*-Methode ersetzt und die Karten (Metal, Wood, Tiger, etc.) sind nun Implementationen des Card-Interface und *CardCategory* wurde entfernt. Somit kann der CardInvalidator in *draw* einfach die invalidate-Methode der gezogenen Karte aufrufen und die Karte führt selbst ihren Effekt aus. Somit können in Zukunft beliebig neue Karten eingeführt werden, für die einfach nur eine Implementation des Card-Interface geschrieben werden muss. Es ist keine Veränderung bestehenden Codes mehr nötig und somit wird das OCP mit dieser Lösung eingehalten.

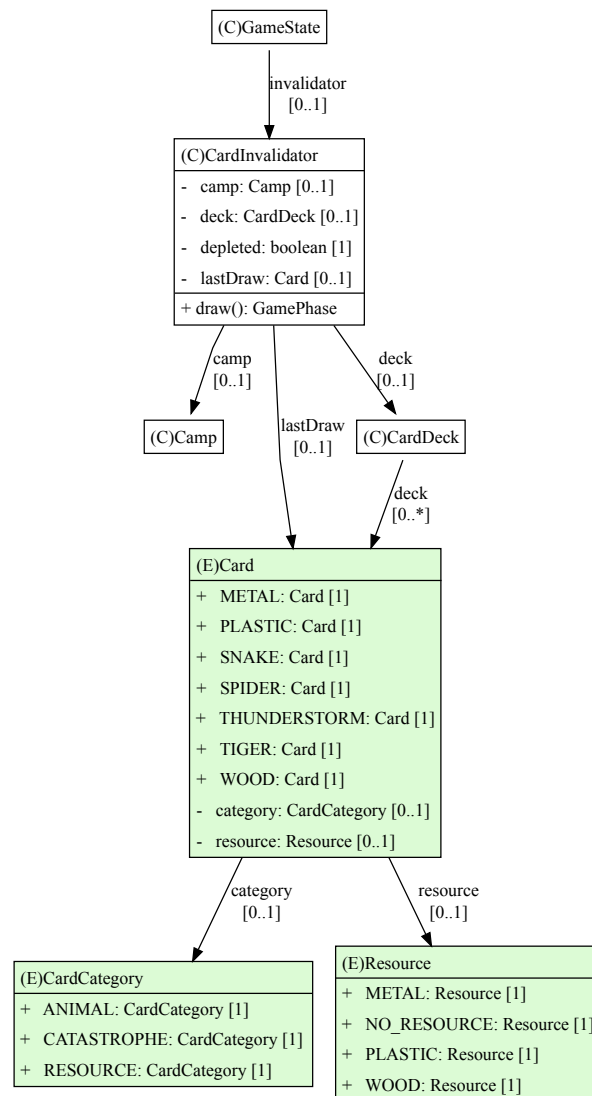


Abbildung 3.5.: UML-Klassendiagramm von *CardInvalidator* und *Card*.

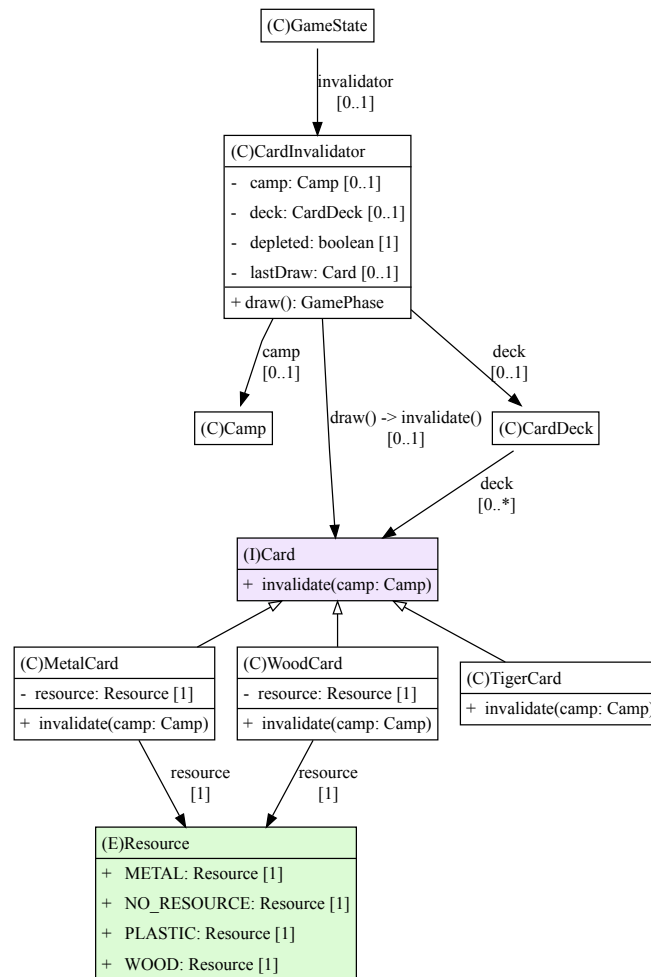


Abbildung 3.6.: Abbildung 3.5 mit Card-Interface und Implementierungen statt Card-Enum und CardCategory. Aus Übersichtlichkeit wurden nicht alle Implementierungen eingezeichnet.

3.3. Analyse Interface-Segregation-Principle (ISP)

Positiv-Beispiel:

Abbildung 3.7 zeigt wie die Interfaces *Buildable*, *Tool*, *Building* und *Rescue* das ISP einhalten. Die Abhängigkeiten von anderen Klassen auf *Tool*, *Building* und *Rescue* sind aus Übersichtlichkeitsgründen ausgelassen. Alle Interfaces enthalten nur eine Methode und alle *Tool*-, *Building*- und *Rescue*-Implementationen sind auch *Buildables* aber nicht

anders herum. So können die Interfaces sehr flexibel angewandt werden und das ISP ist gewahrt.

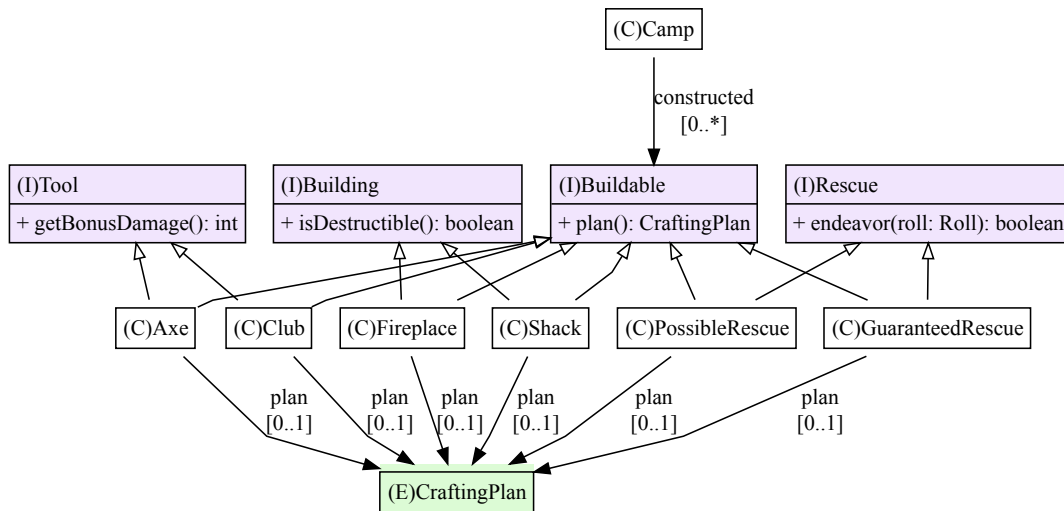


Abbildung 3.7.: UML-Diagramm von Buildable, Endeavor, Tool und Building und deren Implementierungen.

Negativ-Beispiel:

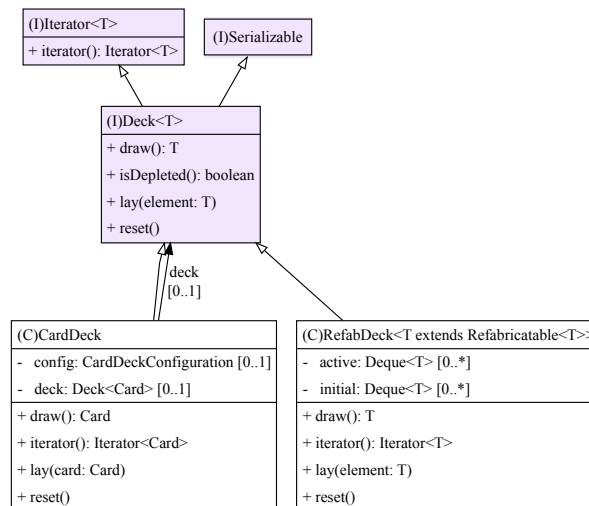
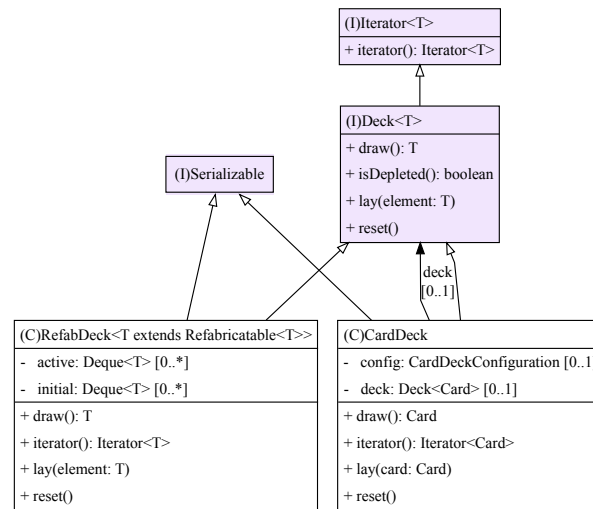


Abbildung 3.8.: UML-Diagramm von *Deck* und dessen Implementierungen.

Abbildung 3.8 zeigt das *Deck*-Interface - das einzige Interface in der Applikation mit mehr als einer Methode. Zunächst ist es allerdings diskutabel, ob es sich hierbei wirklich um ein Negativ-Beispiel handelt, da es sich bei dem Deck-Interface um eine Datenstruktur handelt, die besondere Eigenschaften aufweisen soll, wie z.B. dass der ursprüngliche Zustand der Datenstruktur bei Aufruf der *reset*-Methode wieder hergestellt werden soll. Alle Methoden des Deck-Interfaces werden also gesammelt und gemeinsam benötigt, damit eine Klasse von der Abstraktion des Deck-Interfaces abhängen kann und den nötigen Funktionsumfang verwenden kann. Hier kann also erstmal das Deck-Interface nicht aufgeteilt werden. Außerdem hängt das Deck-Interface von dem *Iterator*-Interface ab, sodass alle Implementationen von Deck auch automatisch Iterator implementieren müssen. Diese Funktion ist aber ebenfalls für ein Deck gewünscht - es soll möglich sein, mit einer Schleife über die Elemente des Deck zu iterieren. Damit ist also auch nicht das Iterator-Interface abspaltbar.

Allerdings wird es unmittelbar klar, dass das ISP hier eindeutig und ohne Diskussion gebrochen wird, wenn betrachtet wird, dass Deck ebenfalls das *Serializable*-Interface beinhaltet, wodurch automatisch alle Implementationen das Serializable-Interface implementieren, was natürlich absolut *keinen* Sinn ergibt. Es ist sogar gefährlich: Beim Implementieren des Deck-Interface wird dem Programmierer nicht klar, dass er die Implementation implizit und ohne Kenntnis als Serializable markiert, was ein großes Problem darstellen kann.

Abbildung 3.9 zeigt nun, dass dieses Problem einfach behoben werden kann, indem die Implementationen explizit und direkt Serializable implementieren und so bewusst die jeweilige Implementation als Serializable markieren. Damit ist das ISP wieder gewahrt, da das Deck so weit wie sinnvoll möglich (Erläuterung s.o.) verkleinert wurde und die Implementationen nun von mehreren kleineren Interfaces abhängen, anstelle von einem größeren abhängen.

Abbildung 3.9.: Abbildung 3.8 mit abgespaltetem *Serializable*-Interface.

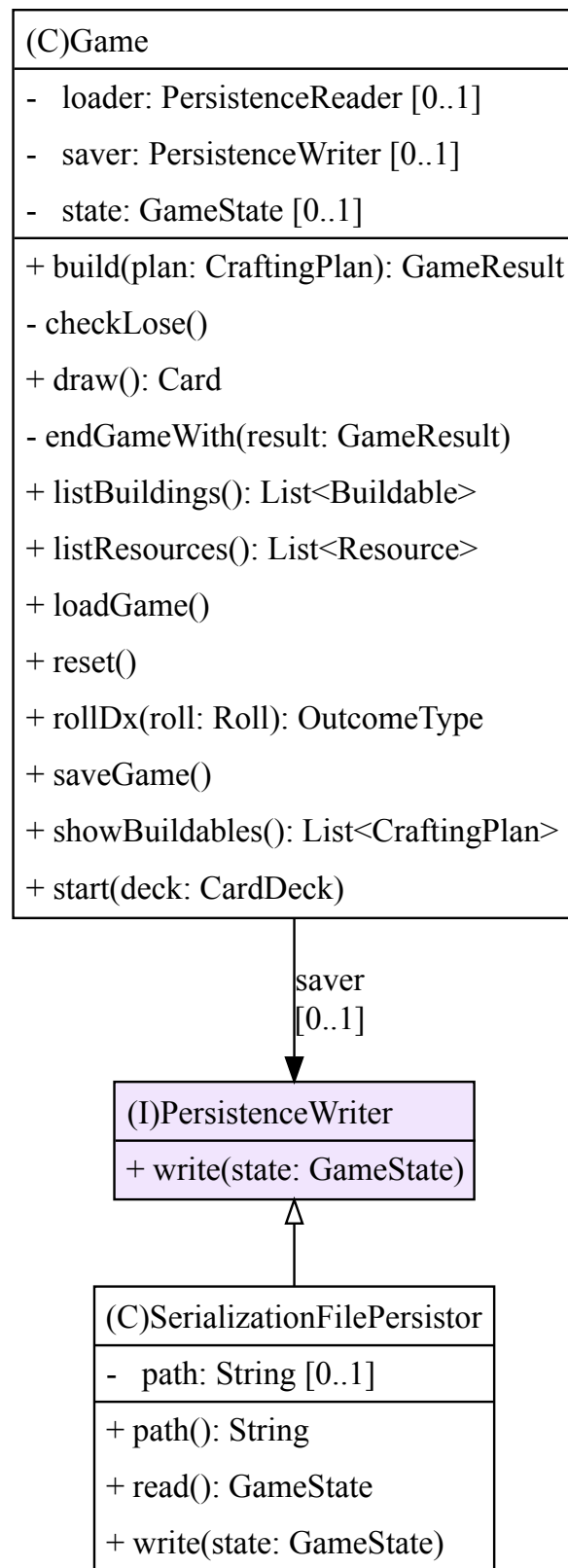
4. Weitere Prinzipien

4.1. Analyse GRASP: Geringe Kopplung

Positiv-Beispiel:

Abbildung 4.1 zeigt das Positiv-Beispiel zur geringen Kopplung.

Das Interface *ase.application.PersistenceWriter* entkoppelt *ase.application.Game* von *ase.plugin.localpersistence.SerializationFilePersistor*. In *Game* soll es die Möglichkeit geben den aktuellen Spielstand zu speichern, damit dieser in Zukunft wieder geladen werden kann und das Spiel fortgesetzt werden kann. *SerializationFilePersistor* bietet diese Möglichkeit, in dem es den aktuellen Spielstand als Java-Objekt serialisiert und als Datei im lokalen Dateisystem speichert. Es darf aber auf keinen Fall eine direkte Abhängigkeit von *Game* zu *SerializationFilePersistor* bestehen, da dies die Dependency Rule brechen würde und außerdem zu einer starken Kopplung führen würde. Mit der Abstraktion durch das Interface ist es möglich in Zukunft andere Speichermethoden einzuführen, ohne dass sich etwas für *Game* ändert, wodurch sich die positiven Effekte der geringen Kopplung entfalten.

Abbildung 4.1.: UML-Diagramm von *ase.application.PersistenceWriter*.

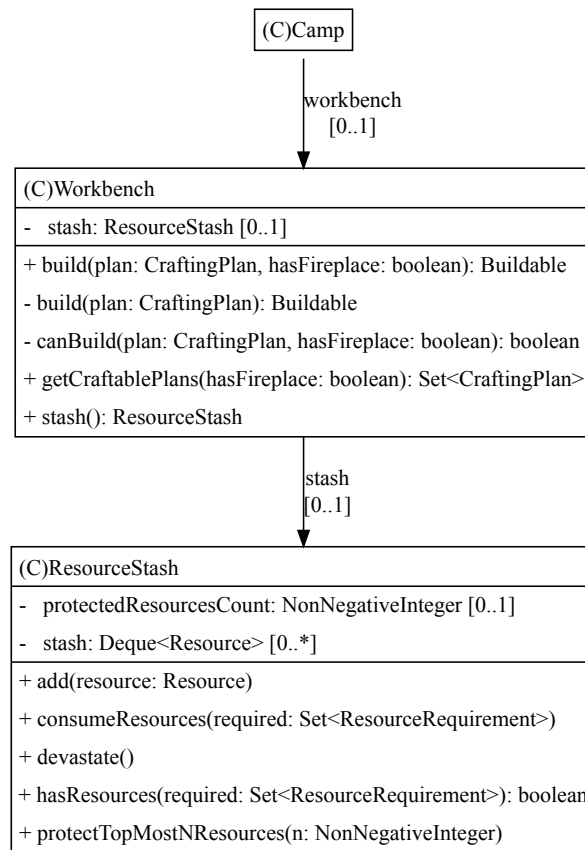
Negativ-Beispiel:

Abbildung 4.2 zeigt das Negativ-Beispiel, da eine sehr starke Kopplung zwischen *Workbench* und *ResourceStash* als direkte Abhängigkeit vorliegt¹.

Aufgabe der *Workbench* ist es zu überprüfen, welche *CraftingPlans* mit den vorhandenen Ressourcen im *ResourceStash* herstellbar sind und Gegenstände mit der *build*-Methode herzustellen, indem dafür die nötigen Ressourcen vom *ResourceStash* verwendet werden. Aufgabe des *ResourceStash* ist es in erster Linie eine Datenstruktur zur Lagerung von Ressourcen bereitzustellen. Die *Workbench* nutzt dabei nur die Methoden *consumeResources* und *hasResources* des *ResourceStash*.

Wie Abbildung 4.3 zeigt, kann die hohe Kopplung einfach aufgelöst werden, indem zwischen *Workbench* und *ResourceStash* ein Interface eingezogen wird, welches die Methoden bereitstellt, die *Workbench* zum verrichten seiner Aufgaben benötigt. Dadurch könnte in Zukunft *ResourceStash* mit einer anderen Datenstruktur ausgetauscht werden, ohne dass die *Workbench* davon etwas mitbekommen würde, aufgrund der nun geringen Kopplung.

¹Das gleiche Problem besteht auch zwischen *Camp* und *Workbench* aber hier wird nur ein Negativ-Beispiel gewünscht und behoben.

Abbildung 4.2.: UML-Diagramm von *ase.domain.crafting.Workbench*.

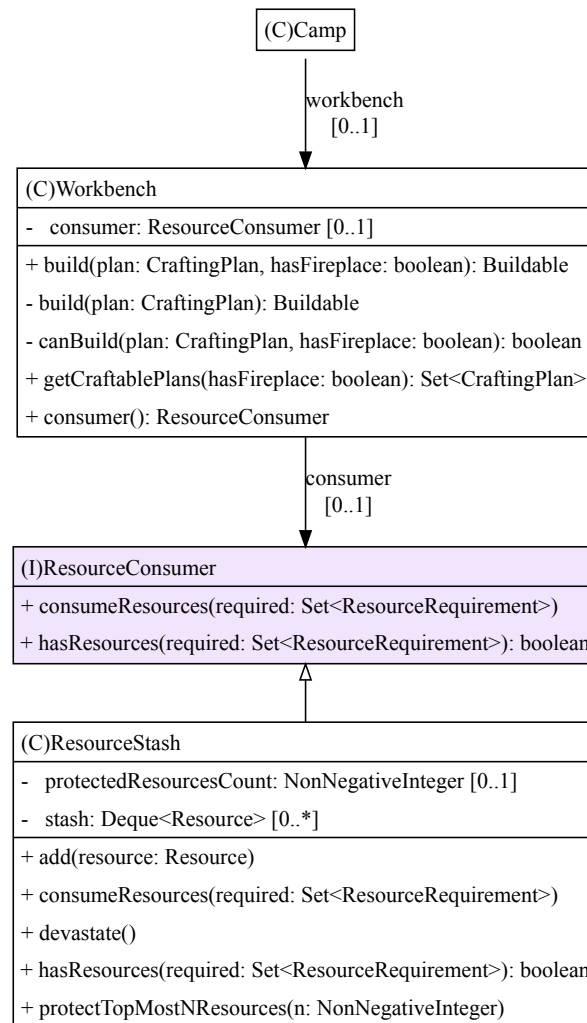


Abbildung 4.3.: Abbildung 4.2 mit eingezogenem *ResourceConsumer*-Interface zur Auflösung der Kopplung.

4.2. Analyse GRASP: Hohe Kohäsion

Abbildung 4.4 zeigt das UML-Diagramm der Klasse *ResourceRequirement*, welches eine sehr hohe Kohäsion hat. Es handelt sich um ein Datentupel aus (Ressource, Menge), welches im *CraftingPlan*-Enum verwendet wird, um die benötigten Ressourcen für das jeweilige Buildable anzugeben. Die Kohäsion ist sehr hoch, da zu einer Bedarfsangabe immer gehört, welcher Gegenstand (*resource*) benötigt wird und wie viel davon (*amount*).

Die beiden Angaben sind im Rahmen der Bedarfsangabe semantisch maximal zusammenhängend und können inhaltlich nicht voneinander getrennt werden.

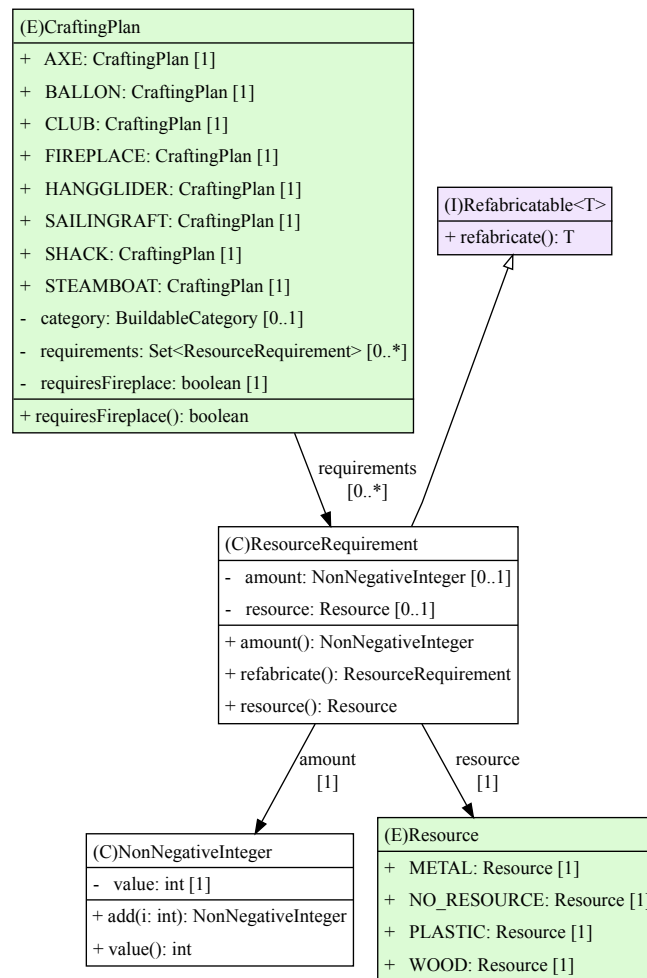


Abbildung 4.4.: UML-Diagramm von *ase.domain.crafting.ResourceRequirement*.

4.3. Don't Repeat Yourself (DRY)

In Commit e246d77b wurde der folgende duplizierte Code aus der *Game*-Klasse gebündelt:

```

state.setPhase(GamePhase.END);
state.setStatus(GameStatus.ENDED);
state.setResult(X);

```

Code-Ausschnitt 4.1 zeigt den Code ein Commit vorher (ID: b4399813). Code-Ausschnitt 4.2 zeigt den Code mit der neuen Methode *endGameWith*, die die das Code-Duplikat an allen drei Stellen entfernt.

Es ist wichtig den Code an der Stelle anzupassen, um bei Änderungen der Game-Schlusslogik, keine Stelle ausversehen zu vergessen, was zu Bugs führen könnte. Ein Beispiel wäre, dass ein zusätzlicher Zustand gesetzt werden muss, Observern ein Statusupdate mitgeteilt werden soll, oder ein anderer Zustand gesetzt werden soll, weil sich das Modell ändert.

Durch das Entfernen der Duplikate ist der Code kürzer, einfacher, besser lesbar (sprechender Name der Methode anstatt willkürlich erscheinende Statusänderungen) und besser wartbar aus den zuvor genannten Gründen.

```
1 // ...
2 // Stelle 1 (Zeile 81-90):
3
4     if (!success) {
5         checkLose();
6         return GameResult.LOSE;
7     }
8
9     state.setPhase(GamePhase.END);
10    state.setStatus(GameStatus.ENDED);
11    state.setResult(GameResult.WIN);
12    return GameResult.WIN;
13 }
14
15 // ...
16 // Stelle 2 (Zeile 106-113):
17
18     if (state.getInvalidator().isDepleted()
19         && !state.getCamp().canBuildAnything()
20         && state.getPhase() != GamePhase.ENCOUNTER) {
21         state.setPhase(GamePhase.END);
22         state.setStatus(GameStatus.ENDED);
23         state.setResult(GameResult.LOSE);
24         return state.getInvalidator().getLastDraw();
```

```
25         }
26
27     // ...
28     // Stelle 3 (Zeile 185-193):
29
30     public void checkLose() {
31         if (state.getInvalidator().isDepleted()
32             && !state.getCamp().canBuildAnything()
33             && state.getPhase() == GamePhase.SCAVENGE) {
34             state.setPhase(GamePhase.END);
35             state.setStatus(GameStatus.ENDED);
36             state.setResult(GameResult.LOSE);
37         }
38     }
39     // ...
```

Code-Ausschnitt 4.1: DRY-Code der *Game*-Klasse zum Commit b4399813.

```
1 // ...
2 // Stelle 1 (Zeile 81-88):
3
4     if (!success) {
5         checkLose();
6         return GameResult.LOSE;
7     }
8
9     endGameWith(GameResult.WIN);
10    return GameResult.WIN;
11 }
12
13 // ...
14 // Stelle 2 (Zeile 104-109):
15
16     if (state.getInvalidator().isDepleted()
17         && !state.getCamp().canBuildAnything()
18         && state.getPhase() != GamePhase.ENCOUNTER) {
19         endGameWith(GameResult.LOSE);
```

```
20         return state.getInvalidator().getLastDraw();
21     }
22
23     // ...
24     // Stelle 3 (Zeile 181-187):
25
26     private void checkLose() {
27         if (state.getInvalidator().isDepleted()
28             && !state.getCamp().canBuildAnything()
29             && state.getPhase() == GamePhase.SCAVENGE) {
30             endGameWith(GameResult.LOSE);
31         }
32     }
33     // ...
34     // Neue Methode (189-193):
35
36     private void endGameWith(GameResult result) {
37         state.setPhase(GamePhase.END);
38         state.setStatus(GameStatus.ENDED);
39         state.setResult(result);
40     }
41     // ...
```

Code-Ausschnitt 4.2: DRY-Code der *Game*-Klasse zum Commit e246d77b.

5. Unit Tests

Mittlere Szenarien

Die Messungen haben ergeben, dass für alle mittelgroßen Szenarien $s \in S_{Anon_M}$ gilt, dass $t_{sub_{MatFlowG}}(s) < t_{sub_{IdObj}}(s)$. Hierbei haben sich für Identical Objects die Laufzeiten über die 13 Szenarien von S_{Anon_M} zwischen 11s und 1min 39s bewegt, während die Laufzeiten von Material Flow Graph zwischen 1s und 9s lagen. Mit $p = 2,744 \cdot 10^{-9}$ ist Material Flow Graph *statistisch signifikant* schneller¹.

Szenario	$t_{sub_{IdObj}}$	$t_{sub_{IdMat}}$	$t_{sub_{MatFlowG}}$	$f_{IdObj \rightarrow MatFlowG}$
s_{Krit}	9h 41min 20s	14min 47s	3min 31s	168,15
s_{Ult}	21h	2h 45min	15min	83,3

Tabelle 5.1.: Ergebnisse der Laufzeitmessungen von S_L nach verschiedenen Methoden. Außerdem der Beschleunigungsfaktor f von Identical Objects zu Material Flow Graph. Szenario s_{Ult} wurde auf einem Server durchgeführt (Speicherbedarf zu hoch), s_{Krit} auf dem in ?? beschriebenen Rechner.

S_X	\bar{m}_{pre}	\bar{m}_{post}	f
small	45,11	5,94	7,6
medium	395,82	120,3	3,29
large	4711,24	50,01	94,19

Tabelle 5.2.: Durchschnittliche Knotenanzahl vor der Graphenreduktion \bar{m}_{pre} , nach der Graphenreduktion \bar{m}_{post} und deren mittlerer Reduktionsfaktor f nach Szenario-Klassen.

Abschließend zur Untersuchung der praktischen Tests lässt sich also festhalten, dass die erwarteten Ergebnisse erzielt, sowie die theoretischen Überlegungen praktisch bestätigt werden konnten.

¹Einseitiger Zwei-Stichproben-t-Test mit unterschiedlicher Varianz; Normalverteilung angenommen; $n = 12$; Szenario z mit $t_{sub_{IdObj}}(z) = 1min39s$ und $t_{sub_{MatFlowG}}(z) = 9s$ als Ausreißer von Test exkludiert.

6. Domain Driven Design

7. Refactoring

In diesem Kapitel wird das Vorgehen in dieser Arbeit zusammengefasst, eine kritische Reflexion vorgenommen, in der unter Anderem die Schwierigkeiten des Projekts aufgeführt werden, sowie eine Aussicht auf die Zukunft des Projekts gegeben.

7.1. Zusammenfassung

7.2. Kritische Reflexion

7.3. Zukunftsaussicht

8. Entwurfsmuster

A. Anhang

A.1. Architekturen

A.2. Quelltext-Implementation