HAW HAMBURG

BACHELORTHESIS
Eugen Deutsch

Untersuchung eines alternativen Modularisierungskonzepts für GUI-Frameworks

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Eugen Deutsch

Untersuchung eines alternativen Modularisierungskonzepts für GUI-Frameworks

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Axel Schmolitzky
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 9. Juli 2019

Eugen Deutsch

Thema der Arbeit

Untersuchung eines alternativen Modularisierungskonzepts für GUI-Frameworks

Stichworte

Graphical User Interface (GUI), Objektorientierte Programmierung (OOP), Java, JavaFX, Swing, Vererbung, Komposition

Kurzzusammenfassung

Das Ziel dieser Thesis ist die Vorstellung eines alternativen Modularisierungskonzepts für GUI-Frameworks, das die Größe der beteiligten Klassen reduziert. Die Thesis konzentriert sich auf klassenbasierte objektorientierte Sprachen wie Java. Es wird gezeigt, dass GUI-Frameworks basierend auf Komposition statt auf Vererbung aufgebaut werden können. Die Thesis startet mit einigen grundlegenden Anforderungen an GUI-Frameworks und zeigt, wie diese in Swing und JavaFX gelöst werden. Es wird gezeigt, wie die Lösungen den Nutzer- und Frameworkcode beeinflussen. Diese Analyse bildet die Grundlage zum Vergleich des Konzepts mit den anderen Lösungen. Das Konzept wird anhand seiner Regeln und einigen Mustern definiert. Zudem wird anhand einer prototypischen Implementation des Konzepts eine Bewertung vorgenommen.

Eugen Deutsch

Title of Thesis

Evaluation of an alternative modularization concept for GUI frameworks

Keywords

Graphical User Interface (GUI), Object Oriented Programming (OOP), Java, JavaFX, Swing, Inheritance, Composition

Abstract

This thesis aims to present an alternative modularization concept for GUI frameworks that reduces the size of the involved classes. The thesis focuses on class based object oriented languages such as java. It shows that a GUI framework can be built on composition instead of inheritance. The thesis starts with by showing some basic requirements for GUI frameworks and how Swing and JavaFX meet them. It shows for each solution how it influences the code of the user and the code of the framework. This analysis will be the base to compare the other solutions with the concept of this thesis. The concept will be defined by a ruleset and some patterns and evaluated by showing a prototypical implementation of it.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 Motivation

In objektorientierten, auf Klassen basierenden Sprachen bilden Klassen eine Einheit zur Aufteilung des Softwareprojekts. Für die Struktur der Klassen muss bedacht werden, dass diese für Eigenschaften wie die Testbarkeit, Lesbarkeit und Erweiterbarkeit unterstützend wirkt. Beispielsweise lässt sich eine Klasse einfacher testen, wenn die Anzahl der Instanzvariablen und somit der Zustand klein ist. Das ergibt sich unter anderem daraus, dass für die Tests ein bestimmter Zustand herbeigeführt und geprüft werden muss. Ist dieser Zustand groß, erschwert dies beide Schritte.
Ähnliches gilt auch für die Gesamtgröße einer Klasse. So heißt es z.B. in [Mar08], S. 136: „The first rule of classes is that they should be small. The second rule of classes is that they should be smaller than that." Auch gibt es psychologische Gründe, die Klassengröße klein zu halten. [Mil56] deutet darauf hin, dass Menschen einen stark begrenzten Raum für gleichzeitig gehaltene Informationen innerhalb eines Kontextes besitzen.

Für die Größe der Klassen stellt sich Implementationsvererbung als schwieriges Konzept heraus. Diese Art der Vererbung kann den Zustand, die Anzahl der Methoden und den Ablauf erweitern, so dass bei der Ansicht einer Klasse mehr bedacht werden muss, als zunächst sichtbar. Deshalb wird mittlerweile häufig empfohlen, Komposition der Vererbung vorzuziehen (Unter anderem in [GJHV11], S. 27 und [Blo17], S. 87). Ein Vorteil der Komposition gegenüber der Vererbung ist, dass hierbei explizit in der jeweiligen Klasse festgeschrieben ist, wo und wann ein Aufruf stattfindet und an wen dieser geht. Bei der Benutzung der Vererbung zeigt hingegen die jeweilige Klasse nur einen Teil des Bildes.

GUI-Frameworks scheinen hier eine Ausnahme darzustellen. In vielen objektorientierten GUI-Frameworks werden für die GUI-Elemente mehrstufige Vererbungshierarchien mit

großen Klassen angelegt. Das ist ein Resultat der vielen Eigenschaften und Aufgaben, die davon verlangt werden könnten. Ein Textfeld kann z.B. eine Schriftart, Schriftfarbe, Position, Größe, Form, Rand und viele andere Eigenschaften haben, die auch andere Formen haben könnten. Diese Eigenschaften werden in abstrakten Klassen zur Verfügung gestellt und wachsen entsprechend stark an. Daraus folgt die Frage, ob und wie GUI-Frameworks strukturiert werden können, so dass den genannten Empfehlungen folgend die Klassengröße im Framework klein ist.

1.2 Ziel der Arbeit

Es sollen Methoden gezeigt werden, mit denen die Klassen der GUI-Elemente funktionalitätsarm und somit klein gehalten werden können und durch Komposition die Funktionalität üblicher GUI-Elemente erreichen können. Es soll dabei sichergestellt werden, dass Erweiterungen nicht auf Kosten der Klassengröße geschehen. Zudem soll ein Konzept für ein GUI-Framework aus den gezeigten Methoden hervorgehen, das mit Swing und JavaFX in den zunächst vorgestellten Kategorien verglichen wird. Zudem wird bewertet, inwiefern Java für das vorgestellte Konzept geeignet ist.

1.3 Vorgehensweise

Zunächst werden einige typische Anforderungen an GUI-Frameworks aufgelistet und im Einzelnen erklärt, wie sich Swing und JavaFX im Nutzer- und Frameworkcode verhalten. Dabei geht es vor allem um die Punkte, die für die Vorstellung des Konzepts von Bedeutung sind. Der Anspruch auf eine vollständige Analyse besteht hier nicht und würde den Rahmen dieser Arbeit überschreiten.

Im Anschluss wird das Konzept dieser Arbeit vorgestellt. Eröffnet wird dies mit den allgemeinen Regeln. Danach werden zentrale Muster vorgestellt und erläutert, wie und wann diese eingesetzt werden und welche Vor- und Nachteile diese haben.

Im dritten Teil der Arbeit wird die Implementation, beginnend mit der Architektur, vorgestellt. Dazu werden im Anschluss die Umsetzung der wichtigsten Komponenten erläutert, sowie die verwendeten Muster und die Vor- und Nachteile der Umsetzung.

Abgeschlossen wird die Arbeit mit einem Fazit, bestehend aus den Vor- und Nachteilen des Konzepts und der Zusammenfassung.

1.4 Begriffe

In dieser Arbeit wird *GUI* statt der Langform *Graphical User Interface* benutzt. Für die Bezeichnung eines Frameworknutzenden wird der Begriff *Nutzer* verwendet. Der Begriff der *Vererbung* meint in dieser Arbeit meistens die Implementationsvererbung. Die Vererbung von Interfaces, wie sie in Java vorkommt, kann in Ausnahmefällen unter diesen Begriff fallen, sofern der Kontext dies ergibt.

1.5 Rahmen

Die Konzepte dieser Arbeit wurden in Java erarbeitet und werden anhand dessen vorgestellt. Javakenntnisse werden, zusammen mit grundlegenden Kenntnissen von Swing und JavaFX, vorausgesetzt. Letzteres äußert sich darin, dass die Quelltextbeispiele gekürzt und nicht für eine volle Anwendung reichen. Insofern wird z.B. das Wissen vorausgesetzt, dass in JavaFX die Startklasse von `Application` erben muss.
In der Analyse und der Implementation wurde Java in der Version 10 benutzt. Die Beispiele der Arbeit nutzen teilweise neuere Konzepte von Java, wie z.B. die Lambda-Schreibweise oder die automatische Typinferenz durch `var`.
Die Analysen von JavaFX beziehen sich auf die Version 12.0.1.

2 Anforderungen an GUI-Frameworks und deren Umsetzung

Im nachfolgenden Kapitel werden einige grundlegende Anforderungen an GUI-Frameworks und deren Lösungen in Swing und JavaFX aufgelistet. Es wird gezeigt, wie die Lösungen den Nutzercode und den Frameworkcode beeinflussen. Sofern ein Punkt bereits zuvor für eine andere Anforderung galt, wird dieser nicht erneut beschrieben, wodurch die Analyse späterer Anforderungen kürzer ausfallen kann.
Die Ergebnisse werden als Basis für den späteren Vergleich mit der Implementation des Konzepts dieser Arbeit dienen.

2.1 Umsetzung von primitiven geometrischen Formen

Als *primitive geometrische Formen* (nachfolgend auch *Primitive*) werden Basisformen bezeichnet, die von einer Grafikbibliothek angeboten werden, um damit komplexere Formen zu erstellen.
In OpenGL sind dies beispielsweise Punkte, Linien, Dreiecke (sogennante Polygone), Vierecke (genannt Quads), sowie Gruppierungen dieser zur Leistungsverbesserung (zu sehen in [Alf09]).
Java2D aus AWT und Swing bietet hingegen einige zusätzliche Primitive an, wie z.B. einen Kreis, verzichtet aber unter anderem auf Dreiecke. Die volle Funktionalität findet sich in [oraa] und [orab]. Die angebotenen Formen sind somit bibliotheksabhängig.

Bei der Umsetzung der Formen gibt es im Wesentlichen zwei Möglichkeiten: Der *Immediate-* und der *Retained*-Modus. Der Retained-Modus arbeitet mit einer Szene, wohingegen der Immediate-Modus Bild für Bild zeichnet. Im ersten Fall wird eine Szene aufgebaut und erst bei Bedarf aktualisiert. Zum Entfernen wäre hier ein Aufruf nötig, der die Szene zur Veränderung veranlasst (siehe [JK18]).
Im Immediate Modus wird hingegen das Zeichnen aktiv gesteuert, so dass ein Entfernen

der Objekte durch Weglassen des Zeichenaufrufs geschieht.

Zusammengefasst wird im Immediate-Modus festgelegt, wie gezeichnet werden soll und im Retained-Modus, wie die Szene aufgebaut sein soll und wie sie abläuft. In dem Sinne ist ersterer imperativ und letzterer deklarativ.

2.1.1 Swing

In Swing erben alle GUI-Elemente von der abstrakten Klasse `JComponent`. Von dieser Klasse erhalten sie die `paintComponent`-Methode, mit der das Zeichnen primitiver Formen möglich ist. Als Argument erhält sie ein `Graphics`-Objekt, das die jeweiligen Methoden zum Zeichnen primitiver Formen anbietet.
Hier wird somit jedes primitive Objekt mit Hilfe eines Methodenaufrufs aktiv gezeichnet und kontrolliert. Somit arbeitet Swing hier im Immediate-Modus.
Das Element, aus dem die `paintComponent`-Methode stammt, wird hingegen im Retained-Modus gezeichnet und unterliegt somit der Verwaltung von Swing.

Nutzercode

Die Anwendung von `paintComponent` zur Umsetzung eines Primitiven sieht wie folgt aus:

```
new JComponent() {
    @Override
    public void paintComponent(Graphics g) {
        g.fillRect(0, 0, getWidth(), getHeight());
    }
}
```

Listing 2.1: Code zum Zeichnen von Primitiven in Swing

Das Problem hierbei ist, dass weder die Dokumentation noch die Vererbungshierarchie darauf deuten, dass diese Schritte zur Umsetzung nötig sind. So heißt es in der Dokumentation von `JComponent` in [orac] lediglich, dass es eine „infrastructure for painting“ bietet. Die Dokumentation von `paintComponent` spricht zwar davon, dass die Methode für Zeichnungen zuständig ist und zählt die Anforderungen auf, es wird aber nicht erklärt, wofür diese Methode überschrieben werden soll. Außerdem gibt es neben `paintComponent`

mehrere andere Methoden, die „paint“ im Namen tragen, `Graphics` als Argument erhalten und dadurch aus Nutzersicht zur Umsetzung in Frage kommen.
Auch die Vererbungshierarchie bietet keine Unterstützung, da `JComponent` auch als abstrakte Klasse keine abstrakten Methoden anbietet, dessen Überschreibung vom Compiler erzwungen werden. Das `paintComponent` die richtige Methode ist, ergibt sich erst aus externen Tutorials wie z.B. aus [orad] von Oracle.

Beim Überschreiben von `paintComponent` ist zu beachten, dass einige Anforderungen gestellt werden. Das schließt mit ein, dass das gegebene `Graphics`-Objekt vor einer Veränderung kopiert werden sollte. Die Schwierigkeit solcher Anforderungen ist, dass sie nicht erzwungen und somit leicht gebrochen werden können.

Ein weiteres Problem ist, dass Primitive durch Operationen von `Graphics` entstehen. Dadurch sind einerseits Primitive keine Objekte und andererseits ist es als Nutzer nicht möglich, `Graphics` neue Operationen beizufügen. Beides ist durch eigene Klassen lösbar, führt aber dazu, dass die Lösung eines Nutzers sich von Swings Lösung unterscheidet.

Frameworkcode
Da `paintComponent` die zentrale Methode zur Umsetzung eigener Zeichnungen ist und `Graphics` als Klasse in der Signatur festlegt, ist dadurch `Graphics` die zentrale Klasse für Zeichenmethoden und Operationen. Unter Berücksichtigung dieser Aspekte ergeben sich für die Erweiterung der Zeichenoperationen folgende zwei Möglichkeiten:

1. Ersatz von `Graphics` in `paintComponent` durch eine neue Klasse mit den gewünschten Operationen.

2. Hinzufügen der Methoden in `Graphics`.

Das Problem an der ersten Möglichkeit ist, dass die Signatur von `paintComponent` bearbeitet werden müsste. Da `paintComponent` beim Erben überschrieben werden kann, ist davon auszugehen, dass Nutzer diese Methode bereits mit der alten Signatur implementiert haben. Das Ändern der Signatur in der Elternklasse würde die Kompatibilität dazu brechen.
Bei der zweiten Möglichkeit muss zwischen dem Hinzufügen von abstrakten und konkreten Methoden unterschieden werden. Beim Einführen neuer abstrakter Methoden ändern sich die Implementationsanforderungen, wodurch all jener Code bricht, der auf der alten

`Graphics`-Version und dessen Anforderungen basiert. Neue konkrete Methoden, die auf den anderen Methoden von `Graphics` basieren würden, wären nicht unmittelbar kompatibilitätsbrechend. Es besteht aber die Gefahr, dass ein Nutzer die neuen Methoden unter gleichem Namen bereits zuvor implementiert hat. In diesem Fall gibt es folgende zwei Möglichkeiten:

1. Die bestehende Methode des Nutzers hat einen anderen Rückgabetyp als die neue Methode in `Graphics`. Das würde den Quelltext des Nutzers brechen.

2. Die bestehende Methode des Nutzers hat den gleichen Rückgabetyp, aber einen anderen Vertrag als die neue Methode. Das würde die Ausführung des Nutzercodes zwar erlauben, die Annahmen, die basierend auf dem Vertrag getroffen werden könnten, würden hier aber nicht gelten.

In der Entwicklung von Swing wurde zur Erweiterung `Graphics` durch `Graphics2D` ersetzt. `Graphics2D` erbt von `Graphics` und fügt einige zusätzlichen Methoden hinzu. Um die Kompatibilität zu wahren, behalten Methoden, die `Graphics` als Parametertyp festlegen, diesen bei. Stattdessen wird beim Aufruf von Swing eine Instanz einer Implementation von `Graphics2D` als Argument eingegeben, so dass zur Nutzung von `Graphics2D` eine Typumwandlung nötig ist:

```
public void paintComponent(Graphics g) {
    final Graphics2D g2d = (Graphics2D) g;
    // ...
}
```

Listing 2.2: Nutzung von `Graphics2D`

Zu `Graphics` lässt sich noch sagen, dass die Klasse 37 abstrakte Methoden enthält und somit hohe Anforderungen an die Implementation stellt. Insgesamt hat die Klasse 49 Methoden und `Graphics2D` erweitert dies um 34 weitere Methoden[1]. Das bedeutet, dass die Erweiterung in diesem Bereich die abstrakten Klassen und damit auch die Implementationen vergrößert hat.

2.1.2 JavaFX

In JavaFX wird das Zeichnen der Primitiven im Retained- und im Immediate-Modus unterstützt. Ersteres geschieht dadurch, dass Primitive Formen in Klassen umgesetzt

[1] Methoden, die von `Object` geerbt werden, sind von den Zählungen ausgenommen.

sind, so dass in der Nutzung die Formen der Szene beigefügt und von da aus durch Methoden manipuliert werden. Die Verwaltung übernimmt JavaFX. Der Immediate-Modus wird hingegen durch eine Klasse `Canvas` unterstützt, welche die jeweiligen Operationen, ähnlich wie `Graphics` in Swing, unterstützt.

Nutzercode

In JavaFX sieht der Umgang mit den Formen wie folgt aus:

```
final var rectangle = new Rectangle(0, 0, 200, 150);
rectangle.setFill(Color.BLUE);
final var circle = new Circle(100, 200, 20);
circle.setFill(Color.GREEN);
```

Listing 2.3: Primitive in JavaFX

Die Primitiven werden somit in eigenständigen Klassen angeboten und müssen nicht vom Nutzer umgesetzt werden. Der Umgang mit Canvas sieht hingegen wie folgt aus:

```
final var canvas = new Canvas(100, 100);
final var gc = canvas.getGraphicsContext2D();
gc.setFill(Color.BLUE);
gc.fillRect(10, 10, 50, 50);
```

Listing 2.4: Primitive durch ein Canvas in JavaFX

Somit kann im Falle dessen, dass die Anwendung beispielsweise die aktive Zeichnung benötigt, auf das `Canvas` zurückgegriffen werden. In anderen Fällen, in denen die Formen statischer genutzt werden, sind sie hingegen als Objekte benutzbar.

Die Primtivklassen erben in JavaFX von der abstrakten `Shape`-Klasse und diese erbt von der ebenfalls abstrakten `Node`-Klasse. In der Dokumentation dieser beiden Klassen heißt es: „An application should not extend the Node class directly. Doing so may lead to an UnsupportedOperationException being thrown.“ Das bedeutet, dass das Problem von Swing bzgl. der unzureichenden Dokumentation für die Vererbung hier dadurch umgangen wird, dass die Klassen von der Vererbung abraten. Der Nachteil dieser Einschränkung ist, dass dadurch die Erweiterung durch den Nutzer anders geschehen muss, als es in JavaFX der Fall ist.

Frameworkcode

Da Primitive von `Shape` und `Node` erben, müssen diese beiden abstrakten Klassen bei der Entwicklung beachtet werden. Schwierig daran ist, dass beide Klassen relativ groß sind. So hat `Node` insgesamt 468 Methoden und 63 Instanzvariablen und `Shape` 57 Methoden und neun statische Variablen. Hinzu kommen mehrere innere Klassen und statische Initialisierungsblocks. Sofern eine neue Klasse für eine primitive Form entwickelt wird, ist es vom konkreten Fall abhängig, wie sehr die jeweiligen Zahlen die Entwicklung beeinflussen. Z.B. hat eine private statische Methode in `Node` keinen direkten Einfluss auf die Komplexität der neuen Klasse. Allerdings kann es vorkommen, dass bei der Entwicklung z.B. die Umsetzung einer Funktion in `Node` nachvollzogen werden muss, um auf diese in der neuen Klasse aufzubauen oder sie zu überschreiben. Selbst wenn die private statische Methode nicht zur Umsetzung dieser Funktion in `Node` genutzt wird, steigert sie zumindest die Größe der Klasse, so dass ein kleiner Ausschnitt aus einer größeren Menge an Zeilen gesucht werden soll.

Die Handhabung der Instanzvariablen geschieht intern wie folgt: Sofern eine Instanzvariable ein Objekt halten soll, wird diese meistens zunächst mit `null` belegt. Wenn eine Methode aufgerufen wird, die das dahinterliegende Objekt benötigt, findet eine `null`-Prüfung statt und bei Bedarf wird das Objekt erstellt.
Diese Schritte sind notwendig, weil JavaFX (und auch Swing) nach dem Prinzip arbeitet, viel Funktionalität zur Verfügung zu stellen, von welcher der Nutzer nur einen kleinen Ausschnitt benötigt. Um die benötigte Menge an Speicher zu reduzieren, wird ein Großteil der Funktionalität deaktiviert. Erst bei Bedarf findet eine Aktivierung statt. Der Nachteil an diesem Prinzip ist, dass wesentlich mehr Abfragen benötigt werden. Dies steigert die Komplexität und Fehleranfälligkeit.

Über die einzelnen Primitiven lässt sich sagen, dass sie bzgl. ihrer Fläche individuelle Methoden anbieten. So beinhaltet die Rechteck-Klasse (nachfolgend `Rect`) die Methoden `setX`, `setY`, `setWidth` und `setHeight`, während eine Linie die Methoden `setStartX`, `setStartY`, `setEndX` und `setEndY` bietet. Diese Methoden stammen nicht von abstrakten Klassen und werden somit direkt von den Primitiven angeboten. Bzgl. mancher dieser Methoden gibt es Überschneidungen in den Klassen. So haben sowohl `Rect` als auch `Text` die `setX` und `setY` Methoden und `Arc` und `Ellipse` haben die `centerX`, `centerY`, `radiusX` und `radiusY` Methoden. Damit verbunden sind die dazugehörigen Zustandsvariablen, Getter und Setter. In `Arc` und `Ellipse` entsteht daraus folgende Codeduplikation:

```
private DoubleProperty centerX;
public final void setCenterX(double value) { ... }
public final double getCenterX() { ... }
public final DoubleProperty centerXProperty() { ... }
```

2.2 GUI-Elemente

Der Begriff *GUI-Elemente* oder auch *Elemente* meint im Folgenden die vom GUI-Framework zur Verfügung gestellten visuellen und interaktiven Formen, wie z.B. ein Button oder ein Textfeld. Auch wenn beispielsweise die Sichtbarkeit eines Buttons deaktiviert werden kann, trifft der Begriff auf einen solchen unsichtbaren Button zu. Primitive Formen gelten hingegen nicht als GUI-Elemente, selbst wenn diese in der konkreten Umsetzung diese Eigenschaften erfüllen.

2.2.1 Swing

In Swing werden den einzelnen Elementen durch eine mehrstufige Vererbungshierarchie ihre Eigenschaften und Funktionalität gegeben. Die Basis bilden dabei die abstrakten Klassen `Component`, `Container` und `JContainer`, welche die allgemeinen Eigenschaften enthalten, wie z.B. die Positionierung, Events, das Enthalten anderer Komponenten und Vieles weitere. Von da an setzen die einzelnen Klassen der Elemente an.
Zum Auslesen und zur Manipulation der Eigenschaften werden zahlreiche Getter und Setter in den Elementen angeboten.

Nutzercode

In Swing sieht der Umgang mit den Elementen wie folgt aus:

```
final var label = new JLabel("Text");
label.setPosition(20, 30);
label.setBackground(Color.BLACK);
label.setForeground(Color.CYAN);
label.setOpaque(true);
```

Listing 2.5: Initialisierung eines Labels in Swing

Zunächst wird der Konstruktor des jeweiligen Elements genutzt, um dann im nächsten Schritt durch Anweisungen die Initialisierung vorzunehmen. Das kostet in der Nutzung häufig zumindest eine zusätzliche Zeile für die meisten Elemente, da die Referenz zum Element für weitere Anpassungen in einer Variable gespeichert werden muss.
Problematischer ist, dass im Quelltext die Übersicht bzgl. der Zugehörigkeit der Setter zum Element fehlt, so dass im Beispiel alle Anweisungen auf einer Ebene sind. Gäbe es ein weiteres Element, so müsste der Nutzer durch Zeilentrennungen, eigene Methoden oder Kommentare die Trennung selbst vollziehen, um die Lesbarkeit zu wahren. Ein Gegenbeispiel hierfür ist das Format in HTML, in dem die jeweilige Zugehörigkeit durch Einrückung sichtbar ist:

```
<table border=5>
    <tr>
        <th>Kopfzeile</th>
    </tr>
    <tr>
        <td align="right">Tabellendaten</td>
    </tr>
</table>
```

Listing 2.6: Ebenen in HTML-Code

Die Einrückung ergibt sich daraus, dass die HTML-Elemente meistens aus einem öffnenden und schließenden Teil bestehen.

Bzgl. des Nutzercodes lässt sich noch sagen, dass die Vererbungshierarchie problematisch sein kann. So enthält beispielsweise `Component` die Methode `setForeground`. Demzufolge erhält auch `JLabel` diese Methode und in diesem Fall bezieht sich die Methode auf die Farbe des Textes. Aus Nutzersicht wäre eine Methode `setTextColor` wesentlich passender, würde aber wiederum `setForeground` obsolet machen. Somit kann es vorkommen,

dass durch die Vererbungshierarchie die Nutzung der Elemente schwieriger wird, weil entweder der Name nicht passt oder das Element die geerbte Funktionalität nicht unterstützt.

Bei der Erstellung eigener Elemente ist von Vorteil, dass durch die Vererbung eine große Funktionalität praktisch kostenfrei erlangt wird. Es gibt aber Methoden, wie z.B. `setEnabled`, die aktiv beachtet werden müssen, damit ihre Anwendung einen Effekt hat. Der Vertrag der Methode legt zwar fest, dass die gesetzte Eigenschaft ignoriert werden kann, ideal wäre aber, wenn die Methode in diesem Fall nicht vorhanden wäre.

Frameworkcode
Zunächst lässt sich sagen, dass `Component`, `Container` und `JComponent` zusammen etwa 20.000 Zeilen haben und dies an jedes Element in Swing weitergeben. Damit verbunden sind viele Instanzvariablen, die den Zustand eines jeden Elements vergrößern und viele Methoden. Das Problem daran ist, dass der Blick in eine Klasse eines GUI-Elements nicht genügt, um seine Aufgaben zu sehen. Entsprechend sind die einzelnen Klassen schwieriger zu verstehen, da diese nur einen begrenzten Teil des am Ende erzeugten Objekts ausmachen.
Swing arbeitet in diesem Bereich wie JavaFX bei den Primtiven und stellt seine Funktionalität deaktiviert zur Verfügung, um sie dann erst bei Bedarf zu aktivieren. Dadurch unterliegt Swing hier den gleichen Nachteilen wie JavaFX (siehe 2.1.2).

Ungewöhnlich sind außerdem die Methoden, die von ihrer Sichtbarkeit aus nicht von außen zugänglich sind, aber durch den `AWTAccessor` zugänglich gemacht werden. Die genaue Erklärung des dahinterstehenden Musters würde den Rahmen dieser Arbeit übersteigen. Wichtig ist hierbei vor allem, dass hier die Sichtbarkeit intern teilweise umgangen wird und somit die Erwartung bzgl. der Aufrufquelle gebrochen wird.

2.2.2 JavaFX

Die GUI-Elemente in JavaFX funktionieren nach einem ähnlichen Prinzip wie die aus Swing. Auch sie haben eine mehrstufige Vererbungshierarchie, die hier aus `Node`, `Parent`, `Region` und `Control` besteht.

Nutzercode

Da GUI-Elemente und Primitive mit `Node` dieselbe Basis haben, ist hier die Gleichbehandlung höher als in Swing, so dass die Lesbarkeit erhöht wird und weniger auf die Unterschiede der Primtiven und GUI-Elemente geachtet werden muss. Ein Unterschied zwischen Elementen und Primitiven sind die Methoden zur Einstellung der Fläche. Während z.B. die `Rect`-Klasse x, y, eine Breite und eine Höhe hat und entsprechende Methoden dafür bereitstellt, ist dies bei GUI-Elementen nicht der Fall. Sie erben dafür lediglich die folgenden Methoden aus `Node`: `setLayoutX`, `setTranslateX`, `setPrefWidth`, `setMinWidth`, `setMaxWidth` und die jeweiligen Methoden für y und die Höhe.
Diese Methoden sind dafür gedacht, die Rahmenbedingung bzgl. der Position und der Größe zu setzen. Die tatsächliche Fläche soll dann von einem Layout verwaltet werden (dazu später mehr ab 2.3).
Damit folgt JavaFX hier dem Prinzip der Festlegung eines Regelwerks, anstatt dass aktiv Anpassungen verlangt werden.

Bzgl. der Erweiterungsfähigkeit für den Nutzer gilt hier das Gleiche wie bei den Primitiven: `Node` rät davon ab, davon zu erben und die anderen Klassen im Zusammenhang mit den GUI-Elementen sprechen nicht davon, wie diese beerbt werden könnten.

Frameworkcode

Die höhere Gleichbehandlung der Primitiven und Elemente ist auch ein Vorteil für die Entwicklung des Frameworks, da hier weniger Unterscheidungen nötig sind. Das bedeutet aber auch das die Nachteile bzgl. der Klassengröße von `Node` für die GUI-Elemente gelten. Da hier eine größere Vererbungshierarchie eingesetzt wird, ist hier der Nachteil größer. So erreichen z.B. das Textfeld und seine Elternklassen folgende Zahlen:

Klasse	Methoden	Variablen [2]	Quelltextzeilen
Node	468	63	10.097
Parent	67	44	1.941
Region	176	37	3.635
Control	31	8	964
TextInputControl	88	19	1.688
TextField	14	3	347

Tabelle 2.1: Zahlen zur Vererbungshierarchie eines Textfeldes in JavaFX

[2] Exklusive statische Variablen

Wie auch bei den Primitiven, bietet JavaFX in den GUI-Elementen zum Teil individuelle Methoden. Die `TextIntputControl`-Klasse (eine der Elternklassen von `TextField`) bietet die Methode `setFont` zum Setzen der Schriftart. Im Vergleich zu Swing, wo die gleiche Methode in `JComponent` angeboten wird und somit allen Swing-Elementen zur Verfügung gestellt wird, ist die Umsetzung in JavaFX spezialisierter. Der Nachteil ist, wie auch schon bei den Primitiven, dass es zu Codeduplikation kommt. In diesem Fall hat auch `Labeled`, eine Elternklasse für Buttons, die gleiche Methode.

2.3 Layouts

Zur Positionierung der GUI-Elemente gibt es neben der absoluten Positionierung, bei der für jedes Element die genaue Position angegeben werden muss, die Layouts. Layouts bieten eine bestimmte Anordnung der Elemente an, so dass sich das Framework hierbei um die Positionierung kümmert. Z.B. ist es in vielen Anwendung nötig, Elemente vertikal hintereinander zu positionieren, so dass sich hier ein entsprechendes Layout anbietet, das diese Aufgabe übernimmt.

2.3.1 Swing

In Swing implementieren die Layouts entweder das `LayoutManager1`- oder `LayoutManager2`-Interface und haben somit teilweise die gleiche Schnittstelle. Zusätzliche enthalten die Layouts viele weitere eigene Methoden. Diese Teilung kommt daher zustande, dass Layouts einerseits von anderen Komponenten abstrakt gehandhabt werden müssen und zum anderen eigene Dienstleistungen anbieten.

In Swing wird zwischen GUI-Elementen und Layouts unterschieden. So werden erstere durch die `add`-Methode an andere Elemente angefügt, während letztere durch `setLayout` gesetzt werden.

Nutzercode
Die Anwendung eines Layouts kann wie folgt aussehen:

```
1 final var frame = new JFrame();
2
3 final var content = frame.getContentPane();
4 content.setLayout(new BorderLayout());
5
6 content.add(new JButton("NORTH"), BorderLayout.NORTH);
7 content.add(new JButton("EAST"), BorderLayout.EAST);
```

Listing 2.7: Anwendung eines Layouts in Swing

Ein Layout ist in Swing als Eigenschaften zu verstehen, die ein Element oder ein Panel[3] hat.
Im Beispiel in Zeile 6 und 7 ist zu sehen, dass die Trennung zwischen dem Panel (Zeile 3) und dem Layout unsauber ist. Die `add`-Methode nimmt neben dem `JButton` eine Einstellung für das `BorderLayout` (*Constraint* genannt) als `Object` an. Das Problem daran ist, dass die Methode nicht erzwingen kann, dass die richtige Einstellung genommen wird. Mögliche Fehlerfälle wären die Wahl der `add`-Methode, die keine Layouteinstellungen annimmt, das Weglassen des `setLayout`-Aufrufs, das Überreichen eines Constraints für ein anderes Layout oder die Wahl eines falschen Typen für ein Constraint.
Außerdem ist ein BorderLayout für nur fünf Elemente gedacht. Das Hinzufügen von weiteren Elementen ist dem Kontext nicht sinnvoll, kann aber durch die `add`-Methode nicht verhindert werden.

Frameworkcode

Die Separierung von Layouts und GUI-Elementen in Swing ist aus der Sicht der Frameworkentwicklung ein Nachteil. Dadurch muss die `add`-Methode, wie zuvor erwähnt, die Constraints für das Layout übernehmen, obwohl weder die Methode noch die Klasse der Methode diese benötigen. Zudem werden intern vor der Weiterleitung der Constraints Typprüfungen bzgl. der Art der Constraints und der Art des Layouts durchgeführt. Bei der Weiterleitung der Constraints erhält das Layout außerdem auch das Element, wodurch es sowohl vom Panel oder Element als auch vom Layout gehalten wird.
Daraus folgt, dass die Separierung die Quelltextmenge, Anzahl der Abfragen und somit die Komplexität und die Codeduplikation erhöht.

[3] Ein Panel ist eine Fläche oder Ebene für Elemente und andere Panels, die sich zur Gruppierung eignen. Die Swing-Panels, wie z.B. `JPanel` erben wie auch die Elemente von `JComponent` und haben somit die gleiche Basis.

Der Vorteil der Teilung ist, dass die Layouts im Vergleich zu den Elementen und ihrer Vererbungshierarchie klein sind. Z.B. hat das `BorderLayout` aus 2.7 893 Zeilen während ein `BoxLayout`-Layout 536 Zeilen hat. Außerdem erbt keine der beiden Klassen.

2.3.2 JavaFX

In JavaFX sind Layouts Elemente, die in der Vererbungshierarchie ihre Basis mit anderen GUI-Elementen teilen. Sie werden als *Panes* bereitgestellt, die eine Kombination aus Panels und Layouts darstellen. Dadurch ist hier die Unterscheidung zwischen einem Layout und einem Element geringer als in Swing.

Nutzercode

Die Zusammenführung der Layouts mit den Panels ist aus Nutzersicht vorteilhaft, da die Panes spezialisiertere Methoden anbieten, als die Panels aus Swing. Dadurch wird z.B. das Problem der Elementbegrenzung beim `BorderLayout` aus Swing gelöst, indem das `BorderPane` in JavaFX je eine Methode für jede unterstützte Position anbietet:

```
1 final var pane = new BorderPane();
2 final var button = new Button("Oben");
3
4 BorderPane.setAlignment(button, Pos.CENTER);
5 pane.setTop(button); // statt add eine spezifische Methode,
6                      //  die den Ort vorgibt
7 pane.setBottom(new Button("Unten");
```

Listing 2.8: Anwendung eines Layouts in JavaFX

Wie in Zeile 4 zu sehen, enthalten Elemente Einstellungsmöglichkeiten für die Panes, so dass auch hier die Fehlerquelle bzgl. der Anwendung der falschen Methode besteht. Im Unterschied zu Swing wird hierbei kein `Object` genommen, so dass hierbei die Typsicherheit gewahrt wird.

Frameworkcode

Panes setzen in der Vererbungshierarchie ab `Region` an, so dass ein Großteil der Elternklassen mit denen der Elemente übereinstimmt. Dadurch fallen hier die Nachteile der

Elemente bzgl. der Klassengröße an während die Gleichbehandlung stärker ist.

Auch die Panes sind in ihren Methoden im Vergleich zu Swing spezialisiert und haben dadurch Überschneidungen. Dies kommt unter anderem in `FlowPane`, `HBox` und `VBox` vor, die alle eine Ausrichtung anbieten und dadurch jeweils die gleiche Implementation der folgenden Methoden anbieten: `setAlignment`, `getAlignment`, `getAlignmentInternal` und `getAlignmentProperty`.

2.4 Events

Events, bzw. Ereignisse im Zusammenhang mit GUI-Elementen beziehen sich auf deren Zustandsänderungen. Diese Zustandsänderungen können z.B. die Änderung der Position, das Klicken eines Buttons oder das Einfügen eines Textes in ein Textfeld sein. Für viele dieser Events bieten GUI-Frameworks eine Möglichkeit an, um als direkte Reaktion eine Aktion folgen zu lassen. Nachfolgend geht es um die Umsetzung dieser Möglichkeit.

2.4.1 Swing

Swing bietet in den Elementen mehrere Methoden zum Hinzufügen von Aktionen, die bei Eintreten des jeweiligen Events ausgeführt werden sollen. Vor allem für den Teil der Events, der durch Hardware wie die Maus entsteht, werden abstrakte Klassen und Interfaces angeboten, die als *Listener* bekannt sind. Andere Events, wie z.B. das Ändern des Textes eines Buttons, werden hingegen durch die `addPropertyChangeListener`-Methode angeboten.

Nutzercode
Für alle Events, für die abstrakte Klassen angeboten werden, müssen diese überschrieben und in die jeweilige Listener-Methode gegeben werden:

```
new JButton().addMouseListener(
    new MouseAdapter() {
        @Override
        public void mouseClicked(MouseEvent e) {
            System.out.println("Klick!");
        }
    }
);
```

Listing 2.9: Der `MouseListener` in Swing

Da Swing vor Java 8 erschienen ist, wurde die Registrierung von Events nicht zugunsten der Lambda-Schreibweise optimiert, so dass die `addMouseListener`-Methode einen `MouseListener` annimmt, der fünf Methoden enthält. Damit nicht alle Methoden implementiert werden müssen, wird hier die abstrakte `MouseAdapater`-Klasse angeboten, die dafür sorgt, dass nur die nötigen Methoden überschrieben werden müssen.

Events durch den `addPropertyChangeListener` unterliegen dem Nachteil der fehlenden Typsicherheit. Das folgt daher, dass diese Events nicht in separaten Klassen und den jeweiligen Methodenaufrufen umgesetzt sind, sondern durch einen `String`:

```
new JButton().addPropertyChangeListener(
    TEXT_CHANGED_PROPERTY,
    e -> System.out.println("Textänderung")
);
```

Listing 2.10: Der `PropertyChangeListener` in Swing

Hier ist im Gegensatz zu dem `MouseListener` die Lambda-Schreibweise für den `PropertyChangeListener` möglich.

Framworkcode

Ein GUI-Element, das von `JComponent` erbt, besitzt mindestens dreizehn Methoden zum Hinzufügen unterschiedlicher Listener. Diese Methoden bestehen dabei nicht aus einfachen Zuweisungen der Listener sondern enthalten einige Abfragen. Hinzu kommen die unterschiedlichen Varianten der Anwendung von `addPropertyChangeListener`, der ebenfalls Abfragen erfordert.
Dies steigert die Komplexität der damit verbundenen Klassen.

2.4.2 JavaFX

JavaFX setzt die Events unter dem Konzept der *Properties* um. Properties vereinen in sich eine Eigenschaft und die Fähigkeit Listener für deren Änderung zu registrieren. Zusätzlich bieten Properties viele weitere Operationen an, wie z.B. die Bildung einer Bindung zwischen zwei Properties:

```
final var first = new Button("Rechts");
final var second = new Button("Mitte");
second.disableProperty().bind(right.pressedProperty());
```

Hier wird die Deaktivierung des zweiten Buttons an das Drücken des ersten Buttons gebunden.

Nutzercode
Aus Nutzersicht ist die Einheitlichkeit der angebotenen Properties vorteilhaft. Fast alle Eigenschaften werden darunter angeboten, so dass die Findung des gewünschten Properties einfacher verläuft als in Swing und nur ein Konzept gelernt werden. Zudem wird hier die Lambda-Schreibweise unterstützt.

JavaFX ändert mit den Properties das Arbeitsprinzip im Vergleich zu Swing. In Swing sind die meisten Eigenschaften mit einfachen Gettern und Settern hinterlegt, so dass die Einstellung der Eigenschaften aktiv vom Nutzer getätigt wird.
Mit den Properties unterstützt JavaFX eine deklarative Beschreibung. Statt die Werte direkt zu setzen, wird mit Properties festgelegt, woraus ihr Wert sich bildet. Das ist zunächst eine ungewohnte Sicht im Vergleich zu Swing, ist aber häufig kürzer.

Frameworkcode
Die Properties sind aus Frameworksicht einheitlicher als die Lösung von Swing, benötigen allerdings mehr Objekte und Methoden. Für viele Eigenschaften werden ein Getter für das Property und ein Getter und Setter für die dahinterstehende Eigenschaft angeboten. Das Property enthält ebenfalls oft einen Getter und einen Setter für die Eigenschaft. Da in der Nutzung selten alle Properties benötigt werden und das Halten von allen speicheraufwändig wäre, werden die Properties erst bei Bedarf initialisiert. Dadurch enthalten die Getter und Setter Abfragen, ob bereits das Property initialisiert wurde.

Die Properties selbst sind umfangreich und haben eine mehrstufige Vererbungshierarchie. Ein `DoublePropertyBase`, das in der `xProperty`-Methode aus `Rect` erstellt wird und die x-Koordinate des Rechtecks betrifft, hat z.B. sechs Elternklassen. Im Vergleich zu den Elternklassen der Elementklassen sind diese wesentlich kleiner, die zusätzliche Komplexität der Vererbungshierarchie besteht aber weiterhin.

2.5 Zusammenfassung

Die Hauptprobleme von Swing und JavaFX ist die Größe der Klassen und deren Komplexität. Beides kommt im Wesentlich dadurch zustande, dass der Großteil der Eigenschaften auf wenige Klassen aufgeteilt werden und diese in Form der Vererbung genutzt werden. In den meisten Fällen bieten die Klassen mehr als benötigt wird, so dass viele Eigenschaften deaktiviert und erst bei Benutzung aktiviert werden. Dieser Ablauf ist in vielen Methoden und steigert durch die Abfragen die Komplexität und senkt die Lesbarkeit. Außerdem ist die Vererbungshierarchie unpräzise, so dass in Swing etwas zu viel vererbt wird während JavaFX dies durch Redefinition verhindert und dadurch Codeduplikation begünstigt.

Bzgl. der Zuweisung der Objekteigenschaften lässt sich sagen, dass sie in beiden Frameworks bei mehreren Objekten unübersichtlich wird.
Dieses Problem hatte JavaFX zumindest in der ersten Version nicht. Zu dieser Zeit wurde JavaFX im JavaFX-Script statt Java umgesetzt und konnte wie folgt geschrieben werden:

```
Stage {
    title: "Animated Counter"
    visible: true
    width: 250
    height: 100
    scene: Scene {
        content: [
            Text {
                content: bind "Counter value is {counter}"
                x: 70
                y: 40
            }
        ]
    }
}
```

Listing 2.11: JavaFX-Script Ausschnitt aus [Top10], S. 38

Hier ist die Zugehörigkeit der Eigenschaften auch bei mehreren Elementen erkennbar.

Bei der Erweiterungsfähigkeit ergeben sich in beiden Frameworks Probleme. Das hängt zum Teil damit zusammen, dass beide methodenbasiert arbeiten und dass nachträgliches Beifügen von Methoden in für Vererbung freigegebenen Klassen schwierig ist.

Für die Primitiven, Elemente und Layouts gilt vor allem in Swing, dass diese sehr unterschiedlich behandelt werden. JavaFX verbessert diesen Umstand durch gemeinsame Elternklassen und steigert so die Verständlichkeit im Umgang mit den jeweiligen Objekten.

3 Das Konzept

Dieses Kapitel beschreibt das Konzept zur Umsetzung eines GUI-Frameworks, das sich auf die Lösung der vorgestellten Probleme von Swing und JavaFX konzentriert. Für das Konzept wird ein grundlegendes Regelwerk festgelegt und die elementaren Muster vorgestellt, die zur Umsetzung der Regeln nötig sind.

3.1 Grundlagen

Im Folgenden werden die grundlegenden Regeln des Modularisierungskonzepts vermittelt. Der nachfolgende Text bezieht sich auf das Framework. Das bedeutet, wenn eine Regel für eine Klasse formuliert wird, so ist damit eine Klasse des Frameworks gemeint.

3.1.1 Vererbung

Die Vererbung von Schnittstellen, in Java durch Interfaces realisiert, ist erlaubt. Für die Implementationsvererbung gelten folgende Regeln:

1. Methoden dürfen nicht überschrieben werden.
2. Es dürfen von der Kindklasse keine Methoden hinzugefügt werden.
3. Es dürfen von der Kindklasse keine Instanzvariablen hinzugefügt werden.

Daraus folgt, dass Kindklassen ausschließlich aus Konstruktoren bestehen. Diese drei Regeln sind eine Ableitung von [Ser19].

Somit ist die Implementationsvererbung im Konzept erlaubt, unterliegt aber strengen Regeln, so dass die Probleme bzgl. des Größen- und Zustandswachstums hier stark reduziert werden.

3.1.2 Methoden

Die Methoden der GUI-Komponenten sollen nur innerhalb anderer GUI-Komponenten verwendet werden. So sollen z.B. die Methoden eines Textfeldes nur vom Layout oder dem Fenster aufgerufen werden, auf denen es sich befindet. Zwar ermöglichen viele Sprachen die Erzwingung dieser Beschränkung nicht, allerdings ist ein ähnliches Resultat möglich, sofern die Methoden anderweitig keinen Nutzen bieten. Methoden, die Eigenschaften eines Objektes direkt setzen oder zurückgeben können, bekannt als Getter und Setter, sollen so weit wie möglich vermieden werden.

Alle Methoden sollen polymorph und nicht überschreibbar sein. Zudem sollen alle Methoden Teil der Schnittstelle einer Klasse sein. Somit sind sie immer öffentlich (in Java `public`).

Methoden sollen, soweit es möglich ist, keine Objekte erzeugen.

3.1.3 Konstruktoren

Es gibt zwei Arten von Konstruktoren: Primäre und sekundäre Konstruktoren. Jede Klasse enthält genau einen Primärkonstruktor, in dem die Instanzvariablen ihren Wert zugewiesen bekommen. Meistens sind dies direkte Zuweisungen der Argumente des Konstruktors an die jeweiligen Instanzvariablen. In manchen Fällen können Transformationen stattfinden, in denen ein Argument zur Erzeugung eines Objekts genutzt wird, welches dann einer Instanzvariable zugewiesen wird. Da dies eine starke Kopplung zur Folge hat und das erzeugte Objekt in Tests nicht ausgetauscht werden kann, ist das eine Ausnahme. Sie gilt nur, wenn das erzeugte Objekt zur Erfüllung der Zuständigkeit unabdingbar und ein Test ohne den Austausch möglich ist. Vor der Zuweisung stellt der Primärkonstruktor sicher, dass die gegebenen Objekte keine Null-Referenzen sind. Auf andere Prüfungen, z.B. ob ein Wert sich innerhalb des gewünschten Bereichs befindet, wird verzichtet.

Die Sekundärkonstruktoren sollen die Erzeugung des Objektes vereinfachen. Dazu stellen sie andere Parameter auf und rufen den Primärkonstruktor mit transformierten Argumenten auf. Diese Transformation geschieht stets durch Erzeugung neuer Objekte. Eine Klasse kann beliebig viele Sekundärkonstruktoren enthalten.

Somit enthalten Konstruktoren nur Nullprüfungen, Objekterzeugungen und Zuweisungen.

3.1.4 Klassen

Es gibt zwei Arten von Klassen: Klassen für aktive und Klassen für passive Objekte. Erstere sind beispielsweise Elementklassen, die unter anderem Methoden zur eigenen Zeichnung und für Events vorgeben. Diese Methoden verändern dabei entweder den Zustand des Elementobjekts oder den Zustand der Argumente. Diese Klassen stellen keine oder nur wenige Methoden mit Rückgabewerten zur Verfügung. Aktive Klassen enthalten somit kaum sondierende Methoden.

Passive Klassen verhalten sich dazu entgegengesetzt. Sie enthalten hauptsächlich Methoden mit Rückgabewerten und keinen oder nur wenigen Argumenten. Zudem haben sie wenige mutierende Methoden. Sie bilden somit den Gegensatz zu den aktiven Objekten.

Das bedeutet, aktive Klassen setzen mutierende Dienstleistung um, während passive Klassen sondierende Dienstleistung enthalten.

Passive Klassen werden von Werttypen unterschieden, da diese einen Zustand haben können.

Die Schnittstelle einer jeden Klasse soll vollständig aus Interfaces zusammengesetzt sein. Wenn z.B. eine Klasse fünf konkrete Methoden hat, so stammen all diese Methoden aus einem oder mehreren Interfaces, welche die Klasse implementiert.

3.1.5 Benennung

Der Name der Klasse, die voraussichtlich am häufigsten vom Nutzer genutzt werden würde, soll am kürzesten sein. Z.B. bezieht sich `Area` auf die Fläche eines Rechtecks. `RectArea` wäre als Name präziser, da aber diese Art der Fläche häufig benutzt werden würde, wird der Präfix entfernt.

Bei Dekorierer-Klassen (später mehr dazu beim betroffenen Muster in 3.2.2) wird der Name des dekorierten Typs weggelassen, wenn eine Instanz des Dekorierers niemals ohne

Einsatz der dekorierten Klasse stattfindet. Die Nutzung würde demzufolge immer wie folgt aussehen:

```
new Logging( // dekoriert eine Area
    new FixArea()
);
```

Listing 3.1: Ein Dekorierer mit verkürztem Namen im Einsatz

Im Beispiel ergibt sich der Namenszusatz für `Logging`, da zu sehen ist, dass eine `Area` angenommen wird. Das setzt voraus, dass `Logging` keine Konstruktoren anbietet, die eine Konstruktion ohne `Area` ermöglichen.

3.2 Verwendete Muster

Nachfolgend werden die verwendeten Entwurfsmuster erklärt. Dabei geht es insbesondere um die für das Konzept wichtigen Muster. Die Beschreibungen konzentrieren sich auf GUI-Frameworks und die Rolle der Muster im Konzept. Muster, die aus [GJHV11] bekannt sind, werden vorausgesetzt und im Gegensatz zu den anderen Mustern, nicht erklärt. Es findet hierbei lediglich die Einordnung in das Konzept statt.

3.2.1 Objektkanal

Zweck
Das Muster dient der Verkleinerung der Schnittstelle und des Zustands einer Klasse mit potenziell vielen Eigenschaften. Außerdem begünstigt das Muster die kompatibilitätserhaltende Erweiterung einer Klasse um eine Eigenschaft.

Problemstellung
In GUI-Frameworks haben viele GUI-Elemente eine Größe. Dazu werden oft direkt die Methoden `getWidth`, `setWidth`, `getHeight` und `setHeight` in der Klasse des Elements positioniert. Da GUI-Elemente viele Eigenschaften haben können, führt dieses Vorgehen zu einem großen Zustand und einer hohen Anzahl an Methoden.
Außerdem benötigen oft viele Klassen diese Eigenschaften, so dass zur Vermeidung von Codeduplikation hier häufig Vererbungshierarchien verwendet werden. Das führt zu einem Teil der vorgestellten Nachteile von Swing und JavaFX.

Lösung
Am Beispiel einer Rechteck-Klasse (nachfolgend `Rect`) soll mit den folgenden Schritten gezeigt werden, wie die besagten Methoden `getWidth`, `setWidth`, `getHeight` und `setHeight` daraus entfernt werden können:

1. Einführung eines neuen Interfaces `Size` mit den besagten Methoden[1].

2. Erstellung einer Implementation von `Size`.

3. Die Einführung eines Konstruktors in `Rect`, der `Size` aufnimmt.

4. Anpassung von `Rect`, so dass statt der eigenen Breite und Höhe (oder den dazugehörigen Methoden) die Methoden aus `Size` benutzt werden.

5. Die Entfernung der Methoden aus `Rect`.

Damit verliert `Rect` die Methoden, wodurch eine Manipulation der Größe nur noch durch das `Size`-Objekt möglich ist. Da das `Size`-Objekt durch keine Methoden angeboten wird, muss die Referenz dazu bei Bedarf zwischengespeichert werden:

```
final var size = new SizeImpl(100, 200);
final var rect = new Rect(size);
size.setWidth(300);
```

Listing 3.2: Manipulation eines Objekts durch ein anderes Objekt

Gesamtheitlich betrachtet werden die Methoden somit nicht gelöscht, sondern transferiert. Zusätzlich entstehen neue Konstruktoren, sofern diese nicht schon vorhanden sind. Zu beachten ist, dass häufig mehrere Konstruktoren für eine Eigenschaft sinnvoll sind. So könnte im Beispiel ein weiterer Konstruktor, der zwei Zahlen annimmt, die Benutzung vereinfachen.

Da hier die Wahl der Eigenschaften individuell je Klasse durch Bereitstellung entsprechender Konstruktoren geschieht, besteht hier auch das Problem der Überversorgung nicht.

Ein Nebeneffekt des Musters ist, dass `Size` für `Rect` jetzt eine Erweiterung darstellt. `Size` kann nicht nur als Größe betrachtet werden, sondern als Kanal für eine Menge

[1] Das Muster lässt sich theoretisch mit Vererbung statt Interfaces umsetzen.

von Objekten, welche die Größe über unterschiedliche Regeln bestimmen können. Z.B. kann eine `ScalingSize`-Klasse eingesetzt werden, die ihre Größe über eine bestimmte Zeit hinweg verändert.
Eine andere Möglichkeit ist eine `FileSize`-Klasse, die sich aus einer Datei heraus ablesen kann:

```
new Rect(
    new FileSize(/* Pfad */)
);
```

Listing 3.3: Auslesen einer Größe aus einer Datei

Dies steht vor allem im Kontrast zu Swing und JavaFX. In beiden Frameworks würde die Größe separat ermittelt und durch Setter gesetzt werden. Das Muster hingegen motiviert zur Erzeugung der Objekte mit dem entsprechenden Verhalten.

Anwendbarkeit
Das Muster eignet sich für Klassen, die viele Eigenschaften haben oder bei denen wahrscheinlich nachträglich Eigenschaften hinzukommen werden. Es gibt dabei Einschränkungen, wenn andere Klassen von den zu transferierenden Eigenschaften abhängig sind.
Im Beispiel hat `Size` die Methoden `getWidth`, `setWidth`, `getHeight` und `setHeight` erhalten. Für diese Methoden lassen sich die Schritte erneut durchführen. So könnte eine Klasse `Scalar` mit den Methoden `getValue` und `setValue` zur Repräsentation eines Wertes eingeführt werden.
Trotz Durchführung der Schritte müssten zumindest die Getter in `Size` bleiben und könnten nicht ersetzt werden. Das liegt daran, dass `Rect` direkt von diesen Methoden abhängig ist und sie zur eigenen Zeichnung benötigt. Unter Umständen lassen sich hingegen die Setter entfernen, da `Rect` diese wahrscheinlich nicht braucht.

Eine zusätzliche Beschränkung betrifft die Art der Eigenschaften. Es ist nur möglich, Eigenschaften hinzuzufügen, die zu den bestehenden Methoden passen. So kann ein `Rect` mit einer Zeichenmethode um alle möglichen Eigenschaften rund ums Zeichnen erweitert werden. Events wären damit aber nicht möglich, sofern der Vertrag der Zeichenmethode nicht gebrochen werden soll.

Im Kontext eines Frameworks führt die konsequente Durchführung dieses Musters zu

einer Teilung der Klassen nach Zugehörigkeit. Die eine Gruppe ist für den Ablauf im Framework zuständig und die andere Gruppe ermöglicht den Umgang mit dem Framework. Entsprechend enthält die erste Gruppe nur frameworkrelevante Methoden. Die anderen Methoden sind hingegen optional und nur sichtbar, wenn die entsprechenden Objekte vom Benutzer eingesetzt werden.

Vorteile

Übersichtlichere Klassen: Durch die Verlagerung der Eigenschaften und Methoden verkleinert sich die Schnittstelle der einzelnen Klassen, so dass die Übersicht steigt. Das gilt auch für die Benutzung. Sollte Interesse daran bestehen, den Code der benutzten Objekte zu analysieren, so enthalten diese Objekte größtenteils benutzte und somit für die Anwendung relevante Methoden.

Genauere Parameterangaben: Da die Teilung der Aufgaben der Klassen stärker ist und die Schnittstellen insgesamt kleiner sind, können bei der Angabe der Parameter genauere Aussagen getroffen werden. Wenn z.B. das `Rect` als Parametertyp einer Methode festgelegt wird, so lässt sich daraus ableiten, dass sie ihre Aufgabe ohne Nutzung der `Rect`-Größe durchführt.
Dies steht im Kontrast zu Swing und JavaFX, in denen der Typ eines Elements eine große Menge an Methoden anbietet. Aus der Angabe eines solchen Parametertyps lässt sich nicht ableiten, welche Dienste davon benötigt werden.

Bessere Testbarkeit: Im Beispiel konnten die Methoden des `Rect`-Objekts in `Size` transferiert werden. Auch die jeweiligen Instanzvariablen von `Rect` werden in `Size` verschoben, wodurch der Zustand von `Rect` kleiner wird. Dadurch werden Tests einfacher, da `Size` zum Einen isoliert getestet werden kann und zum Anderen ein Fake[2] von `Size` für die Tests von `Rect` erstellt werden können.

Bessere Erweiterbarkeit: Durch die Verlagerung von Methoden und die Erweiterung durch Konstruktoren kann die Kompatibilität gewahrt werden, da es keine Kollisionen von Konstruktoren bei Vererbung geben kann. Dies steht vor allem dem Problem von Swing und JavaFX entgegen, dass die Einführung neuer Methoden unter Wahrung der

[2] In manchen Fällen werden für die Erfüllung der Dienstleistungen eines Objekts mehrere andere Objekte benötigt. Damit in Tests das zu testende Objekt isoliert, mögliche Fehler der anderen ausgeschlossen und die Aufrufe abgefangen werden können, werden Fake-Objekte benutzt. Diese implementieren die Schnittstelle der auszutauschenden Objekte und sind lediglich dafür zuständig, die Prüfbarkeit der Aufrufe zu gewährleisten (mehr dazu in [Fea11], S. 47-51).

Kompatibilität schwierig ist.
Wie erwähnt, unterliegt dieser Vorteil der Grenze der bereits bestehenden Methoden. Nur wenn eine neue Eigenschaft in die bisher existierenden Methoden eingegliedert werden kann, ist sie als Erweiterung zulässig.

Einfachere Initialisierung: Da dieses Prinzip Konstruktoren für alle Eigenschaften verlangt, ist es dem Nutzer möglich, ein Objekt durch genau einen Aufruf des Konstruktors zu erstellen. In Kombination mit anderen Klassen entsteht daraus eine baumartige Struktur, die HTML oder XML ähnlich sieht:

```
new Window(
    new Rect( // Inhalt des Fensters
        new AreaImpl( // Fläche des Rechtecks
            new PositionImpl(20, 50),
            new SizeImpl(100, 100)
        )
    )
).show();
```

Listing 3.4: Erstellung eines Fensters mit einem Rechteck durch einen Konstruktoraufruf

Der Vorteil daran ist, dass bereits visuell durch die Einrückung die Zugehörigkeit der Elemente sichtbar ist.

Senkt Temporal Coupling[3]: Ein weiterer Vorteil der baumartigen Struktur ist, dass kein Ablauf besteht. Im Beispiel in 3.4 wurde das Fenster mit der show-Methode gestartet. Die Methode kann in diesem Fall nicht mit einer anderen Zeile getauscht werden, da die Kompilierung dadurch fehlschlagen würde.
Dieser Vorteil besteht in einer ablaufbasierten Form nicht:

[3] Temporal Coupling ist die semantische Kopplung von Zeilen, so dass diese in einer bestimmten Reihenfolge zueinander stehen müssen. Somit muss in der Entwicklung und Wartung bekannt sein, dass die Zeilen in der vorliegenden Reihenfolge bleiben müssen (siehe auch [Bug17], S. 72-73)

```
1 final var window = new Window();
2 final var rect = new Rect();
3 rect.setPosition(20, 50);
4 rect.setSize(100, 100);
5 window.add(rect);
6 window.show();
```

Listing 3.5: Ablaufbasierte Erstellung eines Fensters mit einem Rechteck

In dieser Variante kann z.B. die sechste Zeile mit der dritten vertauscht werden, ohne die Kompilierung zu beeinflussen. Ob dieser Tausch valide ist, entscheidet sich dadurch erst zur Laufzeit.

Nachteile

Entfernung zwischen Aktion und Wirkung: Die Aufteilung der Klassen trennt die Aktion von dem eigentlichen Ziel. Wenn beispielsweise eine Methode von Size aufgerufen wird, ist die Bindung ans Rect syntaktisch nicht deutlich:

```
final var size = new SizeImpl(100, 50);
final var window = new Window(
    new Rect(size),
    new Button(
        new SizeImpl(100, 100),
        () -> size.setWidth(100)
    )
);
```

Listing 3.6: Externalisierung der Aktion

Im Beispiel sind die Größe des Rechtecks und das Rechteck nah beieinander. Wenn allerdings dazwischen viele andere Quelltextzeilen sind oder `Size` als Argument in eine Methode oder einen Konstruktor hineingegeben wird, so steigt der Abstand zum Ort des Geschehens. In diesem Fall muss aktiv darauf geachtet werden, durch passende Bezeichnung der Variablen und der Dokumentation für die Übersicht zu sorgen.

Außerdem ist es möglich, dass Kopplungen zwischen Argumenten entstehen. Wenn beispielsweise eine Methode ein `Rect` und seine Größe benötigt, so müssten dies zwei se-

parate Argumente sein, die trotzdem zueinander gehören. Dadurch entsteht eine neue Fehlerquelle.

JavaFX hat zwar ein ähnliches Prinzip mit den Properties, allerdings haben dort die GUI-Elemente Getter für die Properties. Somit wird auf einen Teil der Vorteile verzichtet, weil die Methoden erhalten bleiben. Die Nachteile sind hier allerdings auch geringer.

Kann mehr Ressourcen benötigen: Beim Einsatz des Musters geht die Möglichkeit verloren, primitive Datentypen statt Objekten zu benutzen. In der Problemstellung wurde `Rect` die Breite und Höhe entzogen und durch ein `Size`-Objekt ersetzt. Dieses Objekt ist für das Muster wichtig und kann nicht durch primitive Datentypen ausgetauscht werden. Im Falle dessen, dass `Rect` sich selbst um seine Größe kümmert, ist es hingegen möglich, auf Objekte dafür zu verzichten und direkt die Werte für die Breite und die Höhe im Rechteck zu speichern.
Zusätzlich erhöht sich die Anzahl der Methodenaufrufe. Da `Rect` unter Einsatz des Musters auf `Size` verweist, sind mehr Methodenaufrufe je Ergebnis nötig.
Andererseits wird ein großer Teil der Zustände im Vergleich zu Vererbungshierarchien eingespart, da jede Klasse individuell entscheidet, was benötigt wird. Damit verbunden sinkt auch die Anzahl der Abfragen. Somit ist ein genauer Vergleich der Einbußen schwierig und vom konkreten Fall abhängig. Laut [GP] ist es außerdem möglich, dass Delegationen zur Laufzeit hinwegoptimiert werden.

Kann zu vielen Konstruktoren führen: Die Benutzbarkeit der Klassen steigt durch zusätzliche Konstruktoren an. Welche Konstruktoren sinnvoll sind, hängt vom genauen Anwendungsfall ab. Oft sind Einsparungen an Zwischenobjekten sinnvoll. So kann ein Sekundärkonstruktor zur Erstellung von `Rect` zwei Ganzzahlen annehmen und die Erwähnung von `Size` einsparen. Da Integer keine weiteren Objekte für ihre Erstellung annehmen, steigern sie die Konstruktoranzahl von `Rect` nicht. Komplexere Objekte haben hingegen die Tendenz die Konstruktoranzahl zu steigern, wobei die genaue Anzahl vom konkreten Fall abhängt.

Mehr Komponenten: Durch die Teilung der Klassen entstehen neue Klassen und Interfaces. Dadurch müssen zur Benutzung der entsprechenden Klassen noch weitere Klassen bekannt sein. Im Gegensatz zu Methoden, für die es in vielen Entwicklungsumgebungen eine Vervollständigungsunterstützung gibt, werden mögliche Klassen nicht

angezeigt. Dies wird vor allem erschwert, wenn Interfaces statt Klassen als Parametertypen angegeben werden.

3.2.2 Dekorierer

Zweck

„Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern." ([GJHV11], S. 175). Somit ist das Muster für das Konzept prädestiniert, da hier die Verwendung von Vererbung reduziert wird.

Problemstellung

Es soll für den Fehlerfall die Möglichkeit geschaffen werden, zusätzliche interne Informationen über GUI-Elemente zu erhalten und so ein Debugging zu vereinfachen. Das kann nicht nur den Zustand des Elements betreffen, sondern auch den Ablauf. In Swing ist beispielsweise die `Component`-Klasse zum Debugging in der Lage und vererbt diese Funktionalität weiter. Der Nachteil ist, dass der Ablauf der Klasse damit vermischt ist. Ideal wäre es, wenn das Debugging getrennt vom restlichen Ablauf wäre.

Lösung

Mit Hilfe des Dekorierermusters ist es möglich, eine separate Klasse basierend auf der Schnittstelle des GUI-Elements zu erstellen. So lässt sich dynamisch zur Laufzeit die Funktionalität nachrüsten und beide Klassen können ihren eigenen Aufgabenfeldern nachgehen:

```
new Window(
    new Debugging( // dekoriertes Rechteck
        new Rect()
    ),
    new Rect() // normales Rechteck
).show();
```

Listing 3.7: Debugging-Dekorierer

Anwendbarkeit

Das Dekoriermuster hat ein ähnliches Aufgabenfeld wie das zuvor vorgestellte Objektkanal-Muster. Beide stellen eine Möglichkeit dar, Objekte um Funktionalität zu erweitern und sind als Alternative zu Vererbung gedacht. Der Unterschied ist, dass das Dekorierermuster extern umgesetzt wird, während für das Objektkanal-Muster intern Änderungen durchgeführt werden müssen.
Damit ist das Dekoriermuster geeigneter, Klassen in ihren Aufgaben zu trennen und führt im Vergleich zu weniger Quelltext in der zu erweiternden Klasse. Es lässt sich allerdings nicht immer umsetzen, da ein Dekorierer abhängig von der Schnittstelle des zu dekorierenden Objekts ist. Wenn sich durch die Schnittstelle eine bestimmte Funktionalität nicht umsetzen lässt, hilft das Muster somit nicht.

Vorteile

Vereinfacht Klassen: Durch die starke Trennung der Aufgaben der Klassen, erhöht sich die Übersicht jeder einzelnen Klasse. Im Vergleich zum Objektkanal sind hier außerdem keine zusätzlichen Instanzvariablen oder Konstruktoren nötig, so dass die Anforderungen geringer sind.

Keine Trennung von Wirkung und Aktion: Ein Dekorierer übernimmt das zu erweiternde Objekt und führt dann seine Aufgabe durch. Hier wird die Wirkung von der Aktion somit nicht getrennt.

Nachteile

Wird von der Schnittstelle beschränkt: Ein Dekorierer kann nur auf die Schnittstelle des dekorierten Objekts zugreifen und hat somit keine internen Informationen.

Kann die Lesbarkeit des Nutzercodes verschlechtern: Das Objektkanal-Muster führt, wie erwähnt, zu einer baumartigen Struktur, in der die Zugehörigkeit der Zuweisungen zu den Elementen erkennbar ist. Dekorierer fügen bei der Aufnahme ihres Objekts im Konstruktor eine neue Ebene hinzu, selbst wenn sie in der laufenden Anwendung keine visuellen Effekte haben.
Zu sehen ist dies in 3.7. Hier befindet sich das undekorierte Rechteck auf einer anderen Ebene als das dekorierte, auch wenn sie in der Anwendung auf der gleichen Ebene sind. Bei vielen Dekorierern auf einzelnen Objekten fällt dieser Nachteil stärker aus, so dass

in diesem Fall neue Klassen zur Zusammenfassung erstellt werden sollten. Dann müssen die Klassen allerdings vom Nutzer erstellt werden, da die genaue Kombination individuell ist und nicht vom Framework alle möglichen Kombinationen angeboten werden können.

Führt zu Codeduplikation: Zur Erweiterung der Funktionalität eines Objektes nimmt der Dekorierer dieses an und verweist in den Methoden auf die jeweilige Methode des dekorierten Objekts. Das führt dazu, dass die Methoden des Dekorierers, die keine zusätzlichen Dienstleistungen anbieten, lediglich die Weiterleitung übernehmen, so dass hier Codeduplikation entsteht.
Zur Reduzierung des Problems gibt es mehrere Möglichkeiten. So führt z.B. eine Reduzierung der Schnittstelle, wie sie in dem Konzept betrieben wird, zu weniger Weiterleitungen.
In [Blo17], S. 91 wird außerdem eine Weiterleitungsklasse gezeigt, welche die Weiterleitung für alle Methoden implementiert. Dadurch kann der Dekorierer davon erben und auf die bloße Weiterleitung verzichten. Nachteilig daran ist, dass eine zusätzliche Klasse je Schnittstelle nötig ist und dies bei wenigen Dekorierern nicht lohnenswert ist.
Ansonsten gibt es einige Sprachen, die das Dekorierermuster in der Sprache verankert haben. In Kotlin z.B. lässt sich das Dekorieren beschreiben, so dass die Weiterleitung eines Dekorierers von der Sprache übernommen wird.

Kann mehr Resourcen benötigen: Hier verhält es sich wie im gleichen Nachteil vom Objektkanal.

3.2.3 Filter

Zweck

Die Erweiterung einer Klasse um neue Funktionalität, basierend auf dessen internen Zustand, ohne diesen Zustand in die Schnittstelle zu tragen.

Problemstellung

Es soll eine Klasse erstellt werden, die ein Fenster repräsentiert. Diese Klasse soll konform mit den Regeln des Konzepts und somit klein und erweiterbar sein. Das Problem ist, dass die Erstellung eines Fensters maßgeblich von der verwendeten Bibliothek zur Erstellung des Fensters abhängig ist und diese es an einen Zustand bindet. Wenn beispielsweise zur

Erstellung der Fenster Swing verwendet werden würde, so wäre das `JFrame` zur Manipulation des Fensters nötig.
Das bedeutet, wenn eine minimale Klasse `Window` unter Einsatz von Swing ihre Dienste anbieten würde, so müsste ihre Schnittstelle eine Erweiterung ermöglichen. Diese Erweiterung ist in dem Fall aber nur über das intern verwendete `JFrame` möglich. Wenn nun das `JFrame` weitergegeben werden würde, so würde die Schnittstelle das Geheimnisprinzip verletzen. Zudem könnte erst die Dokumentation mitteilen, dass das angebotene `JFrame` nur das letzte Mittel sein soll.

Lösung
Zur Lösung wird der Datentransfer umgedreht: Anstatt, dass `Window` einen Getter für die Erweiterungen anbietet, sollen die Erweiterungen eine Methode zur Aufnahme des `JFrame`-Objekts anbieten.
Dazu werden die Erweiterungen in einem neuen Interface zusammengefasst, in dem eine Methode das `JFrame` aufnehmen kann. Außerdem muss nun das `Window` mindestens einen Konstruktor für die Aufnahme der Erweiterungen anbieten und sie zum gewünschten Zeitpunkt ausführen:

```
public interface Feature { // Die Erweiterung
    void apply(JFrame frame);
}

public class Window {
    // ...
    public Window(Feature... features) {
        this.frame = new JFrame();
        for (final Feature f : features) {
            f.apply(frame);
        }
    }
}

// Nutzercode:
new Window(frame -> frame.setVisible(true));
// statt:
new Window().jframe().setVisbile(true);
```

Listing 3.8: Erweiterung eines Fensters durch Filter

Der Name *Filter* ergibt sich aus dem Umstand, dass vom `Window` der eigene Zustand hineingegeben wird und von den Erweiterungen „gefiltert“ wird.

Anwendbarkeit

Wie auch beim Objektkanal- und Dekorierer-Muster, ist hier der Zweck die Aufteilung der Klassen als Alternative zur Vererbung. Der Unterschied ist hierbei, dass dieses Muster auf dem internen Zustand arbeitet. Somit sollte es immer dann verwendet werden, wenn eine Klasse potentiell viele Aufgaben bekommen könnte, diese die Möglichkeit der Aufteilung haben sollen und wenn sie an Interna gebunden sind.
Es ist zu beachten, dass das Muster die Beziehung im Vergleich zum Dekorierer-Muster umdreht und dies nicht in jedem Fall natürlich ist. Beispielsweise ist Vorstellung, dass ein Rechteck eine Größe enthält schlüssig, während das auf den umgedrehten Fall nicht zutrifft.

Vorteile

Die Schnittstelle wahrt das Geheimnisprinzip: Die Schnittstelle muss den Zustand nicht preisgeben. Zwar wird der Zustand immer noch hinausgegeben, dies geschieht allerdings im kleineren Rahmen. Somit ist die Hürde für den Missbrauch etwas höher gegenüber einem entsprechenden Getter.

Ermöglicht Komposition bei Zustandsgebundenheit: Durch das Angebot des Konstruktors können neue Klassen erstellt werden, die als kleine Erweiterung fungieren.

Das Kompositummuster wird ermöglicht: Je nachdem, wie mit den Erweiterungen gearbeitet wird, ist das Kompositummuster anwendbar (mehr zum Muster in [GJHV11], S. 239-253).

Herausgabe wird kontrollierbar: Dadurch dass die Ausführung durch das Ziel erfolgt und das `JFrame` aktiv eingegeben wird, ist es möglich den Prozess zu kontrollieren. So könnte `Window` im Beispiel in 3.8 die `Feature`-Objekte beispielsweise auch nicht ausführen.

Nachteile

Das Geheimnisprinzip wird gebrochen: Auch wenn der Rahmen für den Bruch verkleinert wird, so wird immer noch der interne Zustand zur Manipulation freigegeben. Je nachdem, was dieser Zustand ist, könnte dadurch beispielsweise die Invariante des Objekts gebrochen werden.

Beziehungsumkehr nicht immer sinnvoll: Wie erwähnt, ist das Muster in manchen Fällen nicht einsetzbar, weil die Umkehr der Beziehung unnatürlich wäre.

3.2.4 Abstrakte Fabrik

Zweck

Im Kontext der Arbeit lässt sich ein Nachteil des Dekorierermusters umgehen. Eine allgemeine Vorstellung des Musters findet in [GJHV11], S. 107-118 statt.

Problemstellung

Eine Button-Klasse wurde erstellt, die nur für Buttons ohne Text zuständig ist. Das hat den Hintergrund, dass ohnehin eine Klasse für Beschriftungen erstellt werden sollte, da Button nicht das einzige textenthaltende GUI-Element ist und diese Klasse dann für solche Fälle einsetzbar ist. Somit muss ein Dekorierer erstellt werden, der ein bestehendes GUI-Element mit einem Text versorgt.
Wie erwähnt, wird ein Dekorierer durch die Schnittstelle des Dekorierten begrenzt. Dadurch müsste in diesem Fall der Button seine Position und, je nach Ausrichtung, seine Größe als Getter anbieten. Das ist zwar möglich, allerdings ist in diesem Anwendungsfall die Information nur im Moment der Konstruktion nötig. Zudem sind auch andere Fälle denkbar, in denen andere Informationen zum Zeitpunkt der Konstruktion nötig sind. Würde stets die Nachrüstung der Getter gewählt werden, so wäre eine gemeinsame Schnittstelle schwieriger umzusetzen. Zudem wäre die Schnittstelle größer und als Interface würde sie die Hürden für eine Implementation erhöhen.

Lösung

Als Lösung bietet sich das Muster der abstrakten Fabrik an. Auf diese Weise kann eine `Labeled`-Klasse erstellt werden, die im Konstruktor ein Fabrikobjekt zur Erstellung beliebiger Formen und die Fläche dafür annimmt. Auf diese Weise kann `Labeled` die Konstruktion der gegebenen Form selbst übernehmen und hat gleichzeitig Zugriff auf die Fläche der Form:

```
public interface ShapeFact {
    Shape shape(Area area);
}

public class Labeled implements Shape {
    // ...
    public Labeled(String text, ShapeFact factory, Area area) {
        this.shape = factory.shape(area);
        this.label = new Label(text, area.getPosition());
    }
    // ...
}

// Nutzercode:
new Labeled(
    "Hallo",
    new Area(0, 0, 100, 100), // für Labeled...
    area -> new Button(area, someAction) // und den Button
);
```

Listing 3.9: Fabrikmuster zur Unterstützung des Dekorierermusters

Anwendbarkeit

Das Muster kann verwendet werden, wenn ein Dekorierer zur Erstellung die Informationen des dekorierten Objekts benötigt. Im Zusammenhang mit dem Konzept dieser Arbeit, dient das Muster dazu, die Anwendbarkeit der Dekorierer zu erhöhen. Diese werden dadurch beschränkt, dass die Schnittstellen klein gehalten werden und kaum Informationen hinausgeben.

Vorteile

Hält Schnittstelle klein: Unter Einsatz des Musters lassen sich weiterhin Dekorierer verwenden, ohne dass die Anpassung der Schnittstelle nötig ist.

Erhöht Einsatzbereich von Dekorierern.

Nachteile

Das Muster lässt sich brechen: Im vorherigen Beispiel 3.9 ist zu sehen, dass der Nutzer die Erstellung des Buttons hineingibt und dabei die Nutzung der Fläche steuert. Dadurch ist es möglich, auf die Fläche zu verzichten und den Dekorierer zu umgehen.

Ungewohnte Anwendung: Die Weiterleitung eines Arguments an ein anderes Argument unter Einsatz der Lambda-Schreibweise ist ungewohnt und deshalb zunächst schwieriger zu verstehen.

3.2.5 Alias

Das Muster entstammt aus [Ser19].

Zweck

Die Erstellung einer Klasse, welche die Benutzung des Frameworks vereinfacht (nachfolgend *Convenience-Klasse*).

Problemstellung

Für die Vorstellung der abstrakten Fabrik wurde im Beispiel 3.9 mit `Labeled` ein Dekorierer für einen Button erstellt. Der Aufruf von `Labeled` ist dabei relativ ungewöhnlich. Zusätzlich ist die Erstellung eines Buttons mit einem Text anhand dieses Dekorierers umständlich, zumal ein Button mit einem Text ein häufig benutztes Element ist. Hier wäre eine einfachere Erstellung angebracht.

Lösung

Es wird eine neue Klasse `TextButton` erstellt. Diese Klasse soll den folgenden Aufruf ermöglichen:

```
new TextButton(
    "Hallo",
    new Area(0, 0, 100, 100),
    () -> System.out.println("Geklickt")
);
```

Listing 3.10: `TextButton` als Convenience-Klasse

Unter Einhaltung der Regel bzgl. der Vererbung (siehe 3.1.1) soll diese Klasse eine Kindklasse von `Labeled` sein und dessen Konstruktoren aufrufen, um den Button zu erstellen:

```
public class TextButton extends Labeled {
    public TextButton(String text, Area area, Action action) {
        super(
            text,
            area,
            labeledArea -> new Button(labeledArea, someAction)
        );
    }
}
```

Listing 3.11: Die Klasse `TextButton`

Die einzige Aufgabe einer solchen Convenience-Klasse ist für ein Objekt eine komfortablere Erstellungsmöglichkeit zu bieten und dies unter ihrem Namen zu vereinen (daher der Name Alias für das Muster oder auch für diese Art der Klassen). Dazu benutzt die Convenience-Klasse eine bestehende Klasse und erbt davon, um dessen Konstruktoren ausnutzen.

Anwendbarkeit

Das Muster ist immer dann anzuwenden, wenn für die Erstellung eines Objekts lediglich eine Kombination anderer Objekte benötigt werden und es dafür keine Zusammenfassung gibt.

Außerdem bietet es sich als Mittel an, die Anzahl der Konstruktoren einer Klasse durch Verteilung zu reduzieren.

Vorteile

Unterstützt die anderen Muster: Da die anderen Muster Objektkombinationen hervorbringen (Dekorierer, Filter, abstrakte Fabrik) oder die Anzahl der Konstruktoren steigern (Objektkanal), bietet das Muster eine Reduzierung der Nachteile.

Steigert die Nutzbarkeit des Frameworks: Durch die vereinfachte Erstellung der Objekte, verbessert sich die Nutzbarkeit des Frameworks.

Einfache Struktur: Convenience-Klassen bestehen nur aus Konstruktoren und sind dadurch sehr einfach aufgebaut.

Nachteile

Ungewohnter Aufbau: Das eine Klasse nur Konstruktoren anbietet, ist unüblich und muss zunächst vermittelt werden.

Kann zu unnatürlichen Vererbungsbeziehungen führen: In der Implementation hat sich gezeigt, dass Elternklassen aus pragmatischer Sicht gewählt werden und die Erschaffung einer is-a Beziehung beim Einsatz des Musters in den Hintergrund rückt. Konkret hat sich das bei der Umsetzung einer `Grid`-Klasse gezeigt, die von `Column` geerbt hat.

4 Die Implementation

Im folgenden Kapitel wird die Implementation beschrieben. Diese wurde angefertigt, um das Konzept zu prüfen und dessen Effekte zu ergründen. Zu beachten ist, dass die Implementation aufgrund des hohen Umfangs nur als Prototyp dient und keinem vollständigen GUI-Framework entspricht.

4.1 Verwendete Bibliotheken

Um zu untersuchen, ob auch unter Einsatz des Konzepts zwischen verschiedenen Grafikbibliotheken geschaltet werden kann, ähnlich wie es z.B. in JavaFX der Fall ist, werden in der Implementation Swing und OpenGL verwendet.

4.1.1 Swing

Von Swing wird Java2D zur Zeichnung und das JFrame für die Darstellung der Fenster und die Umsetzung der Events genutzt. Die GUI-Elemente von Swing werden nicht verwendet.

4.1.2 Lwjgl

Lwjgl 3 (Lightweight Java Game Library) ist eine Bibliothek für Java, die Zugriff auf verschiedene native APIs gibt. Wichtig ist dabei OpenGL, das in dieser Arbeit verwendet wird. Die Bibliothek dient als eine Art Wrapper, der größtenteils nur Java-Methoden für die APIs zur Verfügung stellt. Mehr dazu in [lwj].
Damit bietet die Bibliothek einen tiefen Eingriff in die Hardware und bietet dadurch eine hohe Flexibilität, ist aber in der Benutzung schwierig. So müssen Ressourcen von Lwjgl beispielsweise manuell verwaltet werden, da sie teilweise außerhalb des Garbage

Collectors von Java liegen.
Lwjgl stellt neben OpenGL noch GLFW zur Verfügung. Mit GLFW lassen sich Fenster auf verschiedenen Plattformen erstellen. Insgesamt werden hier unter anderem Windows, MacOS und Linux unterstützt. Mehr dazu in [glf].

4.1.3 Besonderheiten

In der Umsetzung hat sich gezeigt, dass Swing und Lwjgl nur unter Einschränkung zusammenpassen. Swing bietet eine höhere Abstraktionsebene als Lwjgl. Damit wurden in Swing einige Designentscheidungen getroffen, die es schwierig machen, diese in der Benutzung wieder umzukehren, um damit einer Lösung in Lwjgl näher zu kommen.
Das äußert sich z.B. in der Handhabung der Fenster. In Swing ist die Erstellung eines Fensters nicht blockierend und startet einen Thread. Der Rest der Anwendung läuft im bestehenden Thread weiter. In Lwjgl ist hingegen eine Schleife zur Aufrechterhaltung des Fensters nötig. Eine solche Schleife könnte im demselben Thread zusammen mit mehreren anderen Fenstern und der gesamten Anwendung laufen.
Da in Swing diese Möglichkeit nicht gegeben ist, müsste sich Lwjgl an Swing orientieren und auf mögliche, zusätzliche Flexibilität verzichten.
Das kostet die Endumsetzung zwar Flexibilität, schließt diese aber nicht aus und genügt dem Rahmen dieser Arbeit.

4.1.4 Ablehnung von JavaFX

JavaFX verwendet intern im Prism Modul laut [Cas] DirectX 9, DirectX 11, OpenGL und Java2D und entscheidet sich je nach Verfügbarkeit für eines davon. Entsprechend wäre es möglich, Prism zu benutzen und dort das eigene Konzept umzusetzen. In diesem Fall wäre die Vergleichbarkeit der Implementation des Konzepts mit JavaFX wesentlich einfacher, da beide die gleiche Basis hätten.
In der Umsetzung ist das allerdings unter anderem deswegen schwierig, weil Prism für solche Zwecke ungenügend dokumentiert ist. Diese Einschätzung lässt sich im Githubprojekt von JavaFX unter [jav] überprüfen. Dort lässt sich feststellen, das viele Klassen und Interfaces kaum oder gar nicht dokumentiert sind.

4.2 Architektur

Die Implementation wurde als Zusammensetzung aus vier Modulen geplant, wobei jedes der Module ein eigenes Projekt sein sollte. Sofern ein Modul ein anderes benötigen würde, sollte es als Bibliothek eingebunden werden. Die jeweiligen Module stünden wie folgt im Verhältnis zueinander:

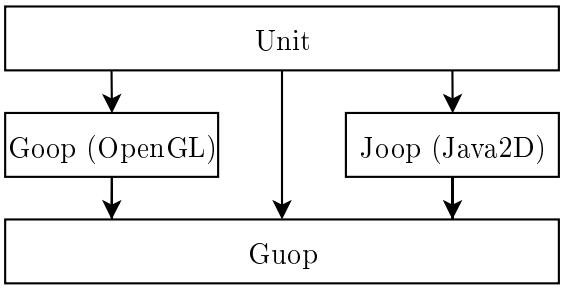

Abbildung 4.1: Beziehungen zwischen den Modulen

Die Aufgaben und die Besonderheiten der Module wären die Folgenden:

Unit: Enthält hauptsächlich Klassen und Interfaces zur Repräsentation von Einheiten, wie z.B. `Size` oder `Pos`. Auch andere gemeinsam genutzte Komponenten werden hier eingefügt. Z.B. sind einige Interfaces zur Erweiterung von `java.util.func` enthalten. Diese würden bei weiterer Entwicklung ein eigenes Projekt erhalten, da aber der zusätzliche Aufwand für die Arbeit keinen Mehrwert bieten würde, entfällt hier die Separierung.

Joop: In Joop werden die Primitive und GUI-Elemente unter Einsatz von Unit zur Verfügung gestellt. Zur Erstellung und Bezeichnung der Fenster werden Swing und Java2D benutzt. Die GUI-Elemente könnten zwar ebenfalls von Swing zur Verfügung gestellt werden, allerdings wird zugunsten der Übersicht und der Performanz darauf verzichtet. So ist das Prinzip des Konzepts die Erstellung eines Minimalfalls, aus dem dann höhere Funktionalität gewonnen werden soll. Ein Swing-Button beispielsweise ist wesentlich funktionsreicher als der Minimalfall, weshalb hier Funktionen deaktiviert werden müssten, um auf diesen zurückzukommen. Außerdem verhält sich ein Swing-Button ähnlich wie ein Fenster und ist nicht ohne Weiteres dekorierbar (dazu später mehr bei der Umsetzung der Fenster in 4.4.4).

Goop: Dieses Modul hat die gleiche Aufgabe wie Joop und verhält sich somit ähnlich. Der Unterschied ist, dass zum Zeichnen OpenGL benutzt wird, wodurch hierbei auch

einige zusätzliche Klassen und Interfaces nötig sind. Es wäre zwar möglich, die sich gleichenden Interfaces in ein separates Projekt zu verschieben und diese dann durch Joop und Goop benutzen zu lassen, allerdings ist die Ähnlichkeit der Strukturen nicht zwingend. Die beiden Projekte könnten ebenso komplett verschiedene Strukturen haben, zumal Java2D und OpenGL unterschiedlich arbeiten. Eine Vorgabe der Struktur durch gemeinsam genutzte Interfaces würde dazu ermutigen, die Struktur gleich zu halten, selbst wenn diese nicht zu Java2D oder OpenGL passen würde.

Guop: Hier werden Joop und Goop zusammengefasst. Je nachdem, ob Java2D oder OpenGL verfügbar ist, wird zwischen den Modulen geschaltet. Somit besteht dieses Modul aus einer einfachen Abfrage und der Weiterleitung an entsprechende Klassen vom jeweiligen Modul. Da sich Joop und Goop strukturell ähneln, schlägt sich das auf die Einfachheit dieses Moduls wider. Allerdings ist dies keine Anforderung. Guop muss mit jeder Art von Asymmetrie umgehen können.

In der ersten Implementation wurden zunächst alle Module unter einem Projekt zusammengefasst. Vor der Umsetzung der Teilung war es möglich, die Zusammenarbeit zwischen den vier Modulen zu gewährleisten. So konnte Guop prüfen, ob OpenGL verfügbar ist und so die Weiterleitung, je nach Ergebnis, an Joop und Goop durchführen. So wurde in einer Fenster-Klasse in Guop die Prüfung durchgeführt, um dann, je nach Verfügbarkeit, an die Fenster-Klasse von Joop oder Goop weiterzuleiten.
Da sich diese Struktur zu replizieren schien und somit zu keinen Erkenntnissen führte, wurde die Entwicklung darin eingestellt. Somit wurde bei der Trennung Unit und Joop umgesetzt, um das Konzept schneller entwickeln zu können.

Die Aufteilung in eigene Projekte, vor allem wenn alle Module umgesetzt werden würden, bringt folgende Vorteile:

Geteilte Nutzung: Die Module können als eigene Bibliotheken eingebunden und genutzt werden. So ist es hierbei z.B. möglich, nur Goop zu benutzen, wenn sicher gestellt ist, dass OpenGL verfügbar ist. Zudem ist es möglich, eigene Frameworks basierend auf den einzelnen Modulen zu schreiben. Dies steht vor allem im Kontrast zu JavaFX, das wie in 4.1.4 erwähnt, eine solche Nutzung schwierig macht.

Kürzere Namen: Das Projekt ist ein Kontext. Wenn beispielsweise Joop benutzt wird, ergibt sich daraus, dass ein verwendetes Fenster auf Java2D basiert. Entsprechend muss dieses keine besonderen Prefix haben und z.B. J2DWindow heißen, so dass in Joop,

Goop und Guop ein Fenster immer Window heißen kann. Dadurch reduziert sich die Länge der Klassennamen, durch die Regel bzgl. der Namensgebung im Konzept in 3.1.5 unterstützt wird.

Höhere Übersicht: Die Gesamtgröße eines Projekts sinkt, was die Übersicht in jedem einzelnen Projekt steigert. Dies ist vor allem von Vorteil, wenn nicht an allen Projekten gearbeitet wird.

Dem entgegen steht der Nachteil, dass Änderungen, die mehrere Module betreffen, umständlicher sind. So muss eine Änderung schrittweise in Abhängigkeitsrichtung durchgeführt werden, wenn die Kompatibilität gewahrt werden soll.
In der Implementation hat sich diese Vorgehensweise trotz des Nachteils als sinnvoll erwiesen hat. Es gibt aber auch Beispiele für große monolithische Projekte. So wird in [PL16] beschrieben, dass Google den Großteil seiner Softwareprojekte in der eigenen Code-Datenbank als einzelnes Projekt hält. Als Vorteile dafür werden unter anderem die Möglichkeit große projektübergreifende Änderungen durchzuführen und die Zusammenarbeit zwischen den Projekten genannt.

In den vorgestellten Modulen sind die wichtigsten Komponenten `Unit`, `Shape`, `Window` und `Event`. Diese stehen in folgender Beziehung zueinander:

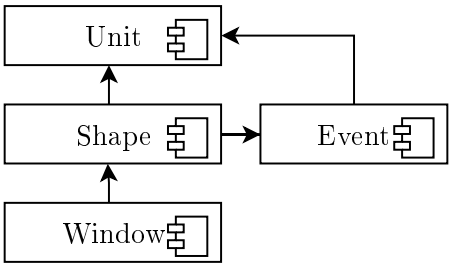

Abbildung 4.2: Beziehungen zwischen den wichtigsten Komponenten

Im Folgenden werden diese jeweils vorgestellt. Dabei werden sie zuerst erklärt, ihre Schwierigkeiten herausgestellt und dann Möglichkeiten aufgezählt, die sie bieten. So soll gezeigt werden, welche Flexibilität geboten wird, ohne dass die Schnittstelle dafür erweitert werden muss. Zu beachten ist dabei, dass `Shape` beispielsweise der Name eines Interfaces ist. Gleichzeitig ist das der Name der Komponente, die Primitive, GUI-Elemente und Layouts unter sich vereint.

4.3 Unit

Die wichtigsten Interfaces des Unit-Moduls sind `Scalar`, `Pos`, `Size` und `Area`. Am Beispiel von `Scalar` wird nachfolgend der Aufbau erklärt:
`Scalar` repräsentiert einen einzelnen Wert innerhalb einer Fläche. So enthalten z.B. die `Pos`-Implementationen zwei `Scalar`-Objekte zur Repräsentation der x- und y-Koordinate. Bis `Area` entsteht damit folgender Aufbau:

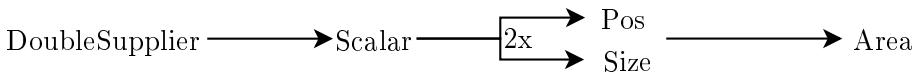

Abbildung 4.3: Interfaces zum Aufbau von Area

`Scalar` hat zwei Aufgaben: Die Übermittlung des eigenen Werts und die Registrierung einer Anpassung für diesen Wert. Für die erste Aufgabe wird dazu ein `DoubleSupplier` aus `java.util.function` in Anspruch genommen. Dieses Interface enthält eine Methode, die einen `double` zurückgibt. Sie wird von der `Scalar`-Implementation aufgerufen, wenn dieser nach dem eigenen Wert gefragt wird. Somit ist folgende Konstruktion möglich:

```
// DoubleSupplier, der System.currentTimeMillis() aufrufen wird
new FixScalar(System::currentTimeMillis);
```

Listing 4.1: DoubleSupplier zur Einführung von Veränderlichkeit

`Scalar` basiert dadurch auf einem sich verändernden Wert. Diese Veränderung lässt sich zudem externalisieren. So lässt sich ein Wrapper `MutableDouble` erstellen, der das `DoubleSupplier`-Interface und zusätzlich eine Methode zum Setzen des eigenen Wertes implementiert:

```
final var scalarValue = new MutableDouble(15); // DoubleSupplier
final var scalar = new FixScalar(scalarValue);
scalarValue.set(30); // Verändert Wert von scalar
```

Listing 4.2: Einführung von Veränderbarkeit

Die Umsetzung mit `Scalar` und `DoubleSupplier` ist die konsequente Durchführung des Objektkanal-Musters aus 3.2.1, wodurch Veränderbarkeit in eine Klasse eingebaut wird, ohne dessen Schnittstelle zu verändern.
Alternativ lassen sich auch andere Implementationen erstellen, die intern die Art der Veränderung enthalten und durch einen Konstruktor die Steuerung dessen ermöglichen:

```
final var mutableBoolean = new MutableBoolean();
new FixScalar(
    new Transitioning(
        new FixScalar(10), // Start
        new FixScalar(50), // Ende
        1_000,             // Nötige Zeit in ms
        mutableBoolean     // Auch hier ein Wrapper, damit
                           //  Changing keine zusätzliche
                           //  Methode braucht
    )
)
```

Listing 4.3: Animation laut Konzept

In diesem Beispiel repräsentiert `Transitioning` den Übergang von einem Wert zum anderen innerhalb einer bestimmten Zeit. Auch hier wird der Start durch ein separates Objekt repräsentiert und hineingegeben. Der Unterschied ist hier, vor allem in Hinsicht auf Swing und JavaFX, dass die Veränderung selbst das Objekt ist. In JavaFX wird beispielsweise die Animation wie folgt umgesetzt:

```
final var label = new Label();
final var translateTransition = new TranslateTransition(
    Duration.millis(2000), label
);
translateTransition.setFromX(0);
translateTransition.setToX(100);
translateTransition.play();
```

Listing 4.4: Animation in JavaFX

Damit hierbei die Animation die Position verändern kann, sind dafür entsprechende Methoden notwendig. Im Konzept wird hingegen nicht die Position extern verändert, sondern verändert sich selbst, wodurch kein Zugang zum Setzen nötig ist.

Die zweite Aufgabe des `Scalars` - das Registrieren einer Anpassung - dient als Kanal für Layouts. Damit ist es Layouts möglich, ihren eigenen Inhalt zu einer bestimmten Anordnung zu verleiten.
Ob ein `Scalar` eine Anpassung annimmt, ist dabei implementationsabhängig. Hier wird

zwischen *soft* und *fix* unterschieden, wobei Erstere Anpassungen annehmen kann und Letztere nicht. So kann z.B. ein Button mit einer „soften“ Breite und einer „fixen“ Höhe erstellt werden, um so nur in einer Richtung anpassbar zu sein.

`Pos`, `Size` und `Area` dienen im Wesentlichen nur zur Weiterleitung an ihre Bestandteile, was letztendlich die `Scalars` sind. So hat `Pos` fünf Methoden: `x`, `y`, `cleanX`, `cleanY` und `adjustment`. Während die ersten beiden Methoden jeweils auf den angepassten Wert des entsprechenden `Scalars` verweisen, werden letztere für den unangepassten Wert benötigt. Diese Unterscheidung hat sich in der Implementation für Layouts als nötig erwiesen.
Das Besondere an den `clean`-Methoden ist dabei, dass sie keinen double zurückgeben, sondern ein `CleanValue`. `CleanValue` ist ein Interface, von dem das `Scalar` Interface die Methoden `cleanValue` und `isFix` erbt. Durch die Teilung der Interfaces ist es möglich, dass `Scalar` direkt zurückzugeben, ohne dass die Methoden von `Pos` umgangen werden können. So wäre z.B. die `adjustment`-Methode von `Pos` nur eine zusätzliche Methode für die gleiche Aufgabe, wenn gleichzeitig ein `Scalar` mit derselben Methode angeboten werden würde. Außerdem würde in diesem Fall die Schnittstelle festschreiben, dass `Scalars` verwendet werden und dass das intern Verwendete verteilt werden muss, da `adjustment` auf einer Kopie das falsche Objekt treffen würde.

Wenn auch in der Implementation nicht so umgesetzt, würden andere Klassen, die als dynamischer Datenträger für GUI-Elemente gelten, diesem Aufbau folgen. Das trifft unter anderem auf `Color` und `Font` zu, so dass die folgenden Möglichkeiten auch auf diese zutreffen würden.

Schwierigkeiten

Mixtur von Fix und Soft: Im Beispiel mit der Animation in 4.3 wurden zwei `RawScalar`-Objekte für `Transitioning` benutzt und dieses wurde wiederum von einem `FixScalar` umhüllt.
Der Hintergrund ist, dass in solchen Fällen für das zusammengesetzte Objekt entschieden werden muss, ob es soft oder fix ist. Dies wird im Beispiel durch die Verwendung von `FixScalar` als Wrapper explizit gemacht, so dass diese Lösung in anderen vergleichbaren Fällen ohne Codeduplikation für die Entscheidung eingesetzt werden kann.
Um diese Entscheidung zu erzwingen, werden alle Schnittstellen, die diese Unterscheidung kennen, aufgeteilt. Dadurch gibt es z.B. `RawPos`, das nur x und y zurückgibt und

`Pos`, welches davon erbt, zusätzlich mit Anpassungen arbeiten kann und somit das Konzept von soft und fix kennt. Für GUI-Elemente wird Letzteres genommen, während Ersteres für besagte kombinierte Objekte genutzt wird.
Durch diese Lösung entsteht eine komplexe Verschachtelung der Interfaces und Klassen, so dass die Nutzung erschwert wird.
Dadurch bedingt ist außerdem, dass z.B. `Pos` nicht auf die `x`- und `y`-Methoden mit `Scalar` als Rückgabe reduziert werden kann. Dies wäre in der Funktionalität gleichwertig und würde die Struktur deutlich vereinfachen. Die Entscheidung zwischen fix und soft wäre hier aber in der Form nicht umsetzbar.

Ständige Abfragen werden vorausgesetzt: Da Veränderung erst durch eine Abfrage erkenntlich wird, ist eine ständige Abfrage notwendig, da sonst Veränderungen verloren gehen. Dadurch werden mehr Zeichenaufrufe durchgeführt und somit mehr Leistung benutzt, die eigentlich nicht benötigt wird.
Bei Flächen und Farben ist es trotzdem möglich auf Abfragen zu setzen. Schwieriger wird es hingegen bei Einstellungen, bei denen die Umsetzung teurer und somit langsamer ist, wie z.B. der Schriftart.
Dieses Problem lässt sich durch Einführung einer Methode zur Registrierung von frameworkinternen Listenern lösen. Ein solcher Listener wäre das Fenster, das sich registrieren würde, um bei Veränderung die Aktualisierung zu veranlassen.
Diese Lösung würde voraussetzen, dass GUI-Elemente, die Unit-Klassen und der von `Scalar` genutzte Wrapper eine solche Methode zur Registrierung hätten.

Möglichkeiten

Veränderung: Veränderung durch Aktionen, Ereignisse, innerhalb einer bestimmten Zeit und somit auch für Animationen.

Listener: Da die Veränderung in einer Methode des genutzten `DoubleSuppliers` stattfindet, muss hier der Mechanismus umgesetzt werden. Die Umsetzung dafür ist aufgrund des kleinen Interfaces einfach.

Bindungen: Vergleichbar mit JavaFX lassen sich Verbindungen zwischen den Unitobjekten knüpfen. So kann der `DoubleSupplier` für einen `Scalar` den Aufruf einer `value`-Methode eines anderen `Scalar`-Objekts enthalten.

Layoutinteraktion: Durch die `adjustment`-Methode.

Begrenzungen: Durch entsprechende Konstruktoren kann unter anderem ein `Scalar` ein `Border`-Objekt annehmen, das als Begrenzung seines eigenen Wertes fungiert. Bei Abfrage des Wertes in `value` kann `Scalar` den Wert vorher durch den `Border` schicken und so, wie bei den Anpassungen, einen veränderten Wert zurückgeben.
Dadurch lässt sich beispielsweise festlegen, dass ein Textfeld nur bis zu einer bestimmten Größe anwächst, wenn ein Fenster sich vergrößert. Auch Gewichtungen, nach denen ein Element mehr von der Größe erhalten soll als ein anderes, sind möglich. Beschränkte Bewegungen lassen sich so ebenfalls umsetzen

Ausrichtung: So konnte eine `Centered`-Klasse implementiert werden, die das `Area`-Interface implementiert und keine anderen Methoden braucht

Zusätzlich gelten die Vor- und Nachteile der Muster aus 3.2.1.

4.4 Shape

`Shape` ist das Interface für alle zeichenbaren Formen und kommt in ähnlicher Form in der Java2D und OpenGL Umsetzung vor. Es schließt Primitive, GUI-Elemente und Layouts mit ein, definiert alleine ihre Schnittstelle und enthält in der aktuellen Implementation drei Methoden:

draw: Sorgt für eine Zeichnung des `Shape`-Objekts. In der Java2D-Version bekommt die Methode dafür das `Graphics`-Objekt. Der Aufruf der Methode geschieht entweder durch das Fenster, von dem aus das Objekt letztendlich kommt, oder durch ein anderes `Shape`, das es enthält

registerFor: Erhält als Argument ein `InputHardware`-Objekt, dass die Registrierung für Maus- und Tastaturevents ermöglicht. Das Objekt wird vom Fenster instantiiert und an die `Shape`-Objekte weitergegeben.

adjustment: Nimmt ein `Adjustment`-Objekt und leitet dieses an `Area` weiter. Wie zuvor erwähnt, wird es letzendlich in `Scalar` registriert und zur Anpassung vom Layout genutzt. Das Besondere an der Methode in `Shape` ist, dass hier zusätzlich noch die `Area`-Referenz zurückgegeben wird. Konkret bedeutet dies, dass wenn ein Layout ein Element anpasst, dieses seine eigene Fläche zurückgibt. Dadurch hat das Layout Zugriff auf die `Scalar`-Objekte in der`CleanValue`-Fassung

Somit ist die Schnittstelle vor allem im Vergleich zu JavaFX und Swing sehr klein. Ein `Shape` kann dennoch eine Vielzahl von Eigenschaften enthalten. Dazu werden, je nach Art, verschiedene Muster verwendet. Nachfolgend wird darauf eingegangen, wie in den einzelnen Bereichen die Erweiterungsfähigkeit gewahrt wird:

4.4.1 Primitive

Umsetzung

Für die Primitiven gibt es wie in JavaFX jeweils eine eigene Klasse. Abseits der unterschiedlichen Zeichnung ist ihre Umsetzung ähnlich: Sie kümmern sich um ihre eigene Zeichnung, indem sie sich durch Einsatz der Farbe und Fläche zeichnen. Zusätzlich registrieren sie beim Aufruf von `registerFor` das gegebene `InputHardware`-Objekt zusammen mit ihrer eigenen Fläche beim Event.
Entsprechend setzt sich der Zustand aus einer Fläche, einer Farbe und einem Event zusammen.

Schwierigkeiten

Viele Klassen: Da jede Form eine eigene Klasse erhält, vergrößert sich die Projektgröße durch zusätzliche Klassen. Dies wird durch die zusätzlichen Klassen zur Unterscheidung von gefüllten und ungefüllten Primitiven verstärkt.

Codeduplikation: Primitive ähneln sich untereinander stark. Erkennbar ist das vor allem in `Rect` und `UnfilledRect`. Der einzige Unterschied dieser Klassen ist die Zeichenoperation, die sie mit `Graphics` ausführen. Somit gleichen die anderen Methoden sich, ebenso wie die Konstruktoren. Hier stellt sich die Frage, ob zumindest die Trennung der Gefüllten und Ungefüllten die richtige Entscheidung war.

Gegenseitige Manipulation möglich: Da die Primitiven das `Graphics`-Objekt vom Fenster benutzen, wirken sich die Änderungen daran auf die anderen `Shape`-Objekte aus.

Möglichkeiten

Alle Fähigkeiten der Bestandteile: Bei den Möglichkeiten der Unitkomponenten in 4.3 hieß es, dass mit einem `Scalar` beispielsweise Bewegung umsetzbar ist. Da die Primitiven stets auf ihre Teile für ihre Aufgaben verweisen, profitieren sie von ihrer Funktionalität. So kann z.B. ein sich verändernder `Scalar` zur Erstellung eines `Rect`-Objekts benutzt werden, so dass beim Zeichnen dieser, sich die Veränderung widerspiegelt. Ähnliches gilt, wenn auch nicht umgesetzt, für `Color` und `Event`.

Neue Primitive: Anhand von `Graphics` lassen sich neue Klassen als Erweiterung erstellen. Dabei sind auch sich verändernde Formen denkbar. Dies steht vor allem entgegen der erwähnten Swingproblematik bzgl. der Erstellung neuer Primitiven.

Layoutinteraktion: Implementiert durch die `adjustment`-Methode.

4.4.2 Elemente

Umsetzung

GUI-Elemente bauen auf Primitive und, wenn nötig, anderen Objekten auf. Da bereits Primitive das `Shape`-Interface erfüllen und die nötige Funktionalität bieten, sind die Klassen für die GUI-Elemente lediglich als einfache Zusammenfassung verschiedener Objekte zu sehen, die sie unter ihrem Namen vereinen. Entsprechend leiten GUI-Elemente in ihren Methoden nur auf die Primitiven weiter.
Die Schwierigkeit bei der Umsetzung ist die minimalistische Schnittstelle. So nimmt z.B. ein Button im Primärkonstruktor ein `Shape` für die eigene Form an. Der Button hat nun mehrere Aufgaben:

1. Möglichst durch die Parametertypen sicherstellen, dass die Fläche vom `Shape` klickbar ist. Eine Linie ist z.B. nicht klickbar.

2. Das Feedback umsetzen. Eine Möglichkeit dafür ist es, eine wechselbare Farbe vom `Shape` zu verlangen, was dann ebenfalls durch den Typ festgelegt werden sollte.

3. Das Event festlegen, bei dessen Erfüllung eine vom Nutzer gegebene Aktion ausgeführt wird. Ein Button besteht dabei aus einem doppelten Event: Es wird beim Klicken und beim Loslassen eine Aktion ausgeführt. Bei Ersterem ist es die Aktivierung des Feedbacks und beim Letzteren die Deaktivierung und die festgelegte Aktion. Somit

muss der Button Zugriff auf die Farbe vom gegebenen Shape haben und ihm das Event geben können.

Diese Aufgaben lassen sich durch das Shape-Interface nicht erfüllen. Da hier allerdings der Zugriff nur zum Konstruktionszeitpunkt erforderlich ist, ist die Umsetzung durch das Muster der abstrakten Fabrik aus 3.2.4 möglich.
Dazu wird statt der Form ein Pen angenommen, der eine Methode zur Erstellung von Shape anbietet. Zusätzlich nimmt der Button eine dazu passende Area, eine wechselbare Farbe und eine Aktion an. Somit erhält der Button Zugriff auf die nötigen Eigenschaften und nutzt sie zur passenden Erstellung eines Shape-Objekts unter Einsatz der Eigenschaften:

```
public <T[1] extends Overlap[2]> Button(
    Pen<T> pen, T area, DualColor color, Action action
) {
    this.shape(
        area,
        color,
        new PressRelease(
            color::toggle,          // Aktion beim Drücken
            () -> {                 // Aktion beim Loslassen
                color.toggle();
                action.run();
            }
        );
    }
}
```

Listing 4.5: Abstrakte Fabrik im Buttonkonstruktor

[1] Durch den generischen Typ T wird erzwungen, dass die gegebene Fläche zu dem Shape passt, das von Pen erzeugt wird.
[2] Overlap ist ein Interface für Flächen, zur Abfrage, ob ein Punkt in der Fläche ist.

Schwierigkeiten

Unüblich: Die Annahme eines Fabrikobjekts zur Erstellung der Form eines GUI-Elements ist unüblich und muss somit für die potentiellen Nutzer vermittelt werden.

Komplexer Primärkonstruktor: Da der Primärkonstruktor Verhältnisse zwischen den Parametern ausdrücken muss (`Area` muss zu `Shape` von `Pen` passen), entsteht mit den Generics ein Konstruktor mit einer vergleichsweise schwierigen Signatur. Dies spiegelt sich auch im Aufruf wider:

```
new Button(
    Rect::new,
    // == (area, color, event) -> new Rect(area, color, event)
    new SoftArea(0, 0, 100, 30),
    new DualColor(new Red(), new Blue()),
    () -> System.out.println("Geklickt")
);
```

Listing 4.6: Primärkonstruktoraufruf eines Buttons

Da der Button auf das Konstruktormuster und die abstrakte Fabrik ansetzt, gelten hier auch dessen Vor- und Nachteile.

Möglichkeiten

Alle Fähigkeiten der Bestandteile: Aus dem gleichen Grund, wie bei den Primitiven.

Design Vorgabe: Mithilfe entsprechender Konstruktoren lassen sich Designs vorgeben. Da im Regelfall weder der Wunsch zur Festlegung der Form noch der Farbe besteht, lässt sich der komplexe Primärkonstruktoraufruf durch Sekundärkonstruktoren verschleiern. So ist auch folgender Aufruf denkbar:

```
new Button(
    new FixArea(0, 0, 100, 30),
    () -> System.out.println("Geklickt")
);
```

Listing 4.7: Sekundärkonstruktoraufruf eines Buttons

Beliebige Formen: Durch die Annahme von `Pen` sind beliebige Formen einsetzbar.

Neue GUI-Elemente: Auch andere GUI-Elemente lassen sich auf die vorgestellte Weise umsetzen. Das gilt auch für komplexere Elemente unter Einsatz bestehender

Elemente. Z.B. enthält die Implementation einen `TextButton`, der sich aus einem Button und einem `Labeled`-Objekt zusammensetzt.

4.4.3 Layouts

Umsetzung

Die Hauptaufgabe der Layouts ist die Anordnung ihres Inhalts mit Hilfe eines `Adjustment`-Objekts über die `adjustment`-Methode. Dieses Objekt enthält vier Methoden, die alle x, y, die Breite und die Höhe annehmen und den jeweiligen angepassten Wert zurückgeben. Für x sieht die Methode wie folgt aus:

```
double adjustedX(
    CleanValue x, CleanValue y, CleanValue w, CleanValue h
);
```

Listing 4.8: Eine der vier Methoden von `Adjustment`

In der Implementation hat sich herausgestellt, dass zur Manipulation einzelner Werte für manche Layouts die volle Fläche nötig ist. Deswegen nehmen die vier Methoden jeweils alle Werte einer Fläche an.

Die `Shape`-Objekte geben das `Adjustment` an ihre `Area` weiter. `Area` teilt es auf und leitet einen Teil an `Pos` und den anderen an `Size` weiter. Auch sie teilen ihr `Adjustment` auf und führen es an ihre `Scalar`-Objekte weiter. Die Aufteilung wird vollzogen, weil der jeweilige `Scalar` nur einen Teil der Fläche repräsentiert und nur dafür die Anpassung umsetzen kann.

Diese Anpassungen basieren auf rechteckigen Flächen und setzen somit voraus, dass beispielsweise auch mehreckige `Shape`-Objekte damit umgehen können:

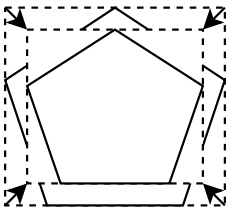

Abbildung 4.4: Rechteckige Verkleinerung eines Pentagons

Im Ablauf geschieht die Anpassung einmal. Es ist auch denkbar, dass ein Layout mehrmals Anpassungen durchführt, z.B. wenn sich dessen Inhalt ändert. Dies ist aber nicht

implementiert.
Bei den Anpassungen kommt es je nach Layout vor, dass die Position der `Shape`-Objekte voneinander abhängig sind. Z.B. gibt es in der Implementation ein Layout `Column`, das seinen Inhalt vertikal abfolgend darstellt, ähnlich wie in JavaFX die `VBox` (mehr dazu in [orae]). Die Position von jedem `Shape` ist somit von der Position des Vorgängers abhängig. Anstatt nun bei jeder Änderung die Anpassungen durchzuführen, wird einmal die Registrierung vollzogen, so dass sich ein jedes `Shape` automatisch selbst korrigiert. Somit wird hier, wie bei den Properties in JavaFX, eine Bindung zwischen den Elementen hergestellt.
Wie bereits bei den Units in 4.3 erwähnt, wird zwischen fix und soft unterschieden, je nachdem ob die Anpassung angenommen wird. Für Layouts ist dieses Wissen vor allem für Größenteilungen wichtig. Wenn beispielsweise ein Layout 500 Pixel in der Breite zur Aufteilung hat und zwei Elemente je Anspruch auf 50 % haben, entstehen Lücken, sobald eines der Elemente seine Breite ablehnt. Zur Vermeidung muss somit das Layout dies berücksichtigen, wofür `Scalar` die Methode `isFix` hat.
Ebenso muss beachtet werden, dass auch Layouts eine fixe oder softe Größe haben könnten und ebenso Anpassungen von anderen Layouts annehmen können müssen. In der Implementation haben sich folgende Regeln herausgebildet, die für Layouts zu beachten sind:

Zyklen sind zu vermeiden: Beim Setzten der Anpassung darf es zu keinen gegenseitigen Abhängigkeiten kommen. Unter diesem Aspekt ist es z.B. nicht erlaubt, dass Layouts ihre Größe von ihrem Inhalt abhängig machen, sofern sie dessen Größe wiederum anhand ihrer eigenen Größe anpassen.
Es ist hingegen erlaubt, dass ein Layout einmal die eigene Größe anhand des Inhalts ermittelt. So ist folgender Aufruf möglich:

```
new Column(
    new Rect(100, 100), // Breite, Höhe
    new Rect(200, 200)
); => w = 200, h = 200
```

Listing 4.9: Layouterstellung ohne Größenangabe

Außerdem dürfen Layouts, wenn sie prozentuale Größenverteilungen vornehmen, sich dabei nur auf den unangepassten Wert beziehen, weshalb die `cleanValue`-Methode in `Scalar` nötig ist.

Ein Layout ist eine Leinwand: Ein `Shape` kann nicht über das Layout hinausgezeich-

net werden. Dies ist zu beachten, wenn ein Shape größer als das Layout ist und dessen Größe fix ist. In diesem Fall behält Shape die angegebene Größe, das Layout bestimmt allerdings die Zeichenfläche und schneidet, bzw. „clipt“ den Überschuss aus. Dies gilt auch für die Events vom Shape.

Größenverhältnisse werden gehalten: Wenn ein Layout zwei Rechtecke enthält und eines davon 20 % der Fläche ausmacht, so sollte bei Vergrößerung dieses Verhältnis aufrecht erhalten werden, sofern beide Größen soft sind. Ein Sonderfall ist hierbei, wenn beide Rechtecke leere Flächen haben. Hierbei würde die Berechnung der Verhältnisse eine Teilung durch null bedeuten, so dass dies abgefangen werden muss.
Die Implementation sieht vor, dass in solchen Fällen eine gleichmäßige Verteilung durchgeführt wird, unter Ausschluss der fixen Größen.

Äußere softe Größen sind dominant: Wenn ein Layout ein Shape enthält, beide eine softe Größe haben und das Layout seine Größe verteilt, so setzt sich die Größe des Layouts durch.

Schwierigkeiten

Schwieriger Ablauf: Das setzen der richtigen Anpassungen hat sich in der Implementation als schwierig herausgestellt. So erschweren hier Abfragen für `isFix`, der Sonderfall mit leeren Flächen und separate Behandlungen von verfügbarer, softer Fläche und unverfügbarer fixer Fläche die Erstellung der Layouts und dessen Verständlichkeit. Außerdem ist hier das Risiko der Zyklenbildung hoch.

Dynamik: In der derzeitigen Implementation sind Formen entweder fix oder soft und bleiben es dann. Somit ist hier das Objektkanal-Muster nicht anwendbar, weil auf nachträgliche Änderungen nicht eingegangen werden würde. Fraglich ist, ob das für jede Anwendung ausreichen würde.
Falls dies unterstützt werden sollte, so müsste die Update-Methode, die bei den Unit-Schwierigkeiten in 4.3 vorgestellt wurde, umgesetzt werden, da die ständige Abfrage unter Umständen zu leistungsintensiv wäre.

Veränderung des Inhalts: Zum Konstruktionszeitpunkt erhält ein Layout seinen Inhalt. Ein Layout bietet keine Methoden zum Tausch, zum Beifügen oder zur Entfernung neuer Elemente. Dies steht auch im Kontrast zu Swing und JavaFX, die Methoden wie `getChildren` in den Elementen anbieten.
Diese Funktionalität ist mit dem Konstruktormuster umsetzbar. Im einfachsten Fall

kann der Nutzer eine Referenz auf die gegebene `Shape`-Liste halten und diese verändern. Die Schwierigkeit ist hier, dass Layouts bei Aktionen auf der Liste die Anpassungen neu vornehmen müssen.

Möglichkeiten

Neue Layouts: Unter der gleichen Schnittstelle lassen sich neue Layouts erstellen. Diese können auch aus anderen Layouts zusammengesetzt sein. In der Implementation existiert z.B. ein einfaches `Grid`-Layout, das lediglich aus einer `Column` unter einer Menge von `Row`-Objekten zusammengesetzt ist.

Listenfunktionen: Zwar nicht umgesetzt und wie in den Schwierigkeiten erklärt, schwierig, allerdings lassen sich auch Listenfunktionen zum Tauschen, Hinzufügen und Entfernen der Elemente beifügen.

Shapefunktionalität: Vorgegeben durch das `Shape`-Interface, können Layouts Events unterstützen und z.B. die Sichtbarkeit durch entsprechende Dekorierer kontrollieren lassen. Außerdem können sie mit anderen Layouts interagieren.

4.4.4 Window

Umsetzung

Das Besondere an einem Fenster ist, dass es von sich aus nicht dekorierbar ist. Das folgt daher, dass die Manipulation von Fenstern an einen bestimmten bibliotheksabhängigen Zustand (*Handle*) gebunden sind. In Lwjgl ist die Rückgabe bei Erzeugung eines Fensters ein `long`, wohingegen in Swing das Fenster an ein `JFrame` gebunden ist.
Um nun einem `Window`-Objekt von außen eine zusätzliche Funktionalität zu geben, müsste der Zugriff auf das interne Handle bestehen, was das Geheimnisprinzip brechen würde.
Zur Umgehung dieses Problems wird das Filtermuster aus 3.2.3 angewandt. Es wird von `Window` mindestens ein Konstruktor angeboten, der ein `Feature` annimmt. Ein `Feature` ist ein Interface mit einer Methode, die ein `JFrame` akzeptiert. In der Benutzung sieht dies wie folgt aus:

```
new Window(
    new Features(
        jframe -> jframe.setTitle("Titel"),
        jframe -> jframe.setSize(300, 300)
    )
);
```

Listing 4.10: Konstruktion eines Fensters mit Features

Dadurch kann die Schnittstelle der Fenster klein gehalten werden. Diese besteht aus Methode `show`, womit das Fenster erstellt und aufgerufen wird.

Schwierigkeiten
Es gelten die Vor- und Nachteile des Filter-Musters.

Möglichkeiten
Durch den Einsatz des Filters ist es möglich, die Schnittstelle des Fensters klein zu halten und die Funktionalität des `JFrame`-Objekts zu nutzen. Das Fenster hat in der Implementation nur die Methode `show` zum starten.
Unter Anwendung des Alias-Musters aus 3.2.5 ist es zudem möglich, den Aufruf mit den `Features` zu verstecken. Z.B. wird in der Implementation eine Fenster-Klasse angeboten, die einen Fenstertitel annimmt und die Erstellung des dafür benötigten `Feature`-Objekts übernimmt.

4.4.5 Events

Umsetzung
Für Events wird ein Interface angeboten, das eine Methode `registerFor` bietet. Diese Methode nimmt ein `InputHardware`- und ein `Overlap`-Objekt an. Damit ähnelt sie der gleichnamigen Methode von `Shape`.
`InputHardware` dient als eine Abstraktion von `MouseListener`, `KeyListener` und zwei anderen Interfaces für Eingabe-Events. Hingegen ist `Overlap` ein Interface mit einer Methode, die bestimmen kann, ob sich ein Punkt in einer Fläche befindet.
In der Anwendung sieht ein Event wie folgt aus:

```
new Window(
    new Rect(
        new FixArea(100, 100),
        new Press(
            () -> System.out.println("Gedrückt")
        )
    )
).show();
```

Listing 4.11: Erstellung eines Events

Bei Ausführung der `show`-Methode wird das Fenster erstellt. Dabei ruft `Window` die Methode `registerFor` in `Rect` mit einem `InputHardware`-Objekt auf. `Rect` leitet dieses Objekt weiter an `Press` und übergibt zusätzlich noch die eigene Fläche.
`Press` nutzt diese Objekte und registriert das richtige Event und die ihm gegebene Aktion am `InputHardware`-Objekt.

Schwierigkeiten

Unterliegen Grenzen ihres Bereichs: Ein Event arbeitet nur in dem Bereich, für den es festgelegt wurde. Dadurch sind übergreifende Events, wie z.B. das Verschieben eines Elements, in der Form schwer umzusetzen.

Speicherverschwendung: Die Event-Objekte sind nach Vollzug ihrer Dienstleitungen für gewöhnlich unbrauchbar und verschwenden Speicherplatz, da ihr `Shape` sie weiterhin hält.

Dynamik: Die Umsetzung von Dynamik ist für Events ähnlich schwierig wie bei den Layouts. Die Herausforderung hier besteht vor allem darin, nachträglich die Deregistrierung durchzuführen.

Möglichkeiten

Komposition: Events lassen sich kombinieren, um neue Events zu erstellen. Die Umsetzung dessen lässt sich in eigenen Klassen durchführen, wodurch kombinierte Events und einfache Events in der Nutzung gleich wirken.

4.4.6 Nutzercode

In der Zusammensetzung kann der Einsatz des Frameworks wie folgt aussehen:

```
final Size size = new SoftSize(100, 100);
final Shape button = new Button(
    size,
    () -> System.out.println("Geklickt!")
);

new BaseWindow( // Argumente: Size, Feature, Shape
    size,
    new Features(
        new Centered(), // Das Fenster soll zentriert...
        new ShapeScaling(button), // ...das Shape skalieren...
        jframe -> jframe.setResizable(true) // ...und selbst
                                            // skalierbar sein
    ),
    button
).show();
```

Listing 4.12: Erstellung eines Fensters mit einem Button

Es wird ein Fenster erstellt, das einen Button enthält. Das Fenster nimmt ein `Feature`-Objekt auf, welches durch das `Features`-Objekt auf drei erweitert wird. Die Features sorgen für die Zentrierung und die Skalierbarkeit des Fensters und das Mitskalieren des Buttons, wobei die ersten beiden Features in eigenen Klassen angeboten werden.
Das Fenster und der Button haben dieselbe Größe und da die Größe Soft ist, akzeptiert diese die Skalierung des Fensters.

Da in diesem Beispiel einige typische Anwendungsfälle angewandt werden, können Convenience-Klassen benutzt werden, um den Code zu vereinfachen:

```
new ScaleWindow(
  new Button(
    new SoftSize(100, 100),
    () -> System.out.println("Geklickt!")
  )
).show();
```

Listing 4.13: Erstellung eines Fensters mit einem Button

`ScaleWindow` erbt von `Window`, welches wiederum von `BaseWindow` erbt und ist eine Convenience-Klasse, die den oberen Anwendungsfall abdeckt.

Aus Nutzersicht ergeben sich im Vergleich zu Swing und JavaFX einige Vorteile. Darunter fällt, dass alle Klassen stets Interfaces als Parametertypen angeben und die Interfaces klein sind, so dass hier die Erweiterung aufgrund der niedrigeren Implementationsanforderungen und Komplexität einfacher ist.
Außerdem ist es aus Nutzersicht einfacher, den Ablauf des eigenen Codes intern für eigene Erweiterungen zu analysieren, da die genutzten Objekte festlegen, welche Menge an Quelltext im Hintergrund abläuft.

Die Zugehörigkeit der Eigenschaftszuweisungen wird hier getrennt, da die Konstruktoren immer alle Eigenschaften aufnehmen und ihren Bereich der Zuweisung zur Verfügung stellen, ähnlich wie im JavaFX1-Beispiel aus 2.11.

Der Nachteil ist, dass der Nutzer die jeweils zu nutzenden Objekte und die zugehörigen Konstruktoren kennen muss. Außerdem ist die Struktur sehr ungewöhnlich, so dass hier die Nachvollziehbarkeit erschwert wird.

5 Fazit

Zu den Vorteilen des vorgestellten Konzepts gehört vor allem die starke Verkleinerung der Klassen und Schnittstellen. Primitive, Elemente und Layouts konnten in der Implementation unter einer gemeinsamen Schnittstelle, die aus drei Methoden besteht, vereint werden. Die Menge an Instanzvariablen in den Klassen konnte ebenso reduziert werden. Zudem zeigt das Konzept Vorteile im Bereich der Erweiterbarkeit, Testbarkeit und der Einheitlichkeit. Die Ziele dieser Arbeit wurden somit in den vorgestellten Bereichen erreicht. Die Kosten dessen sind die Folgenden:
Das vorgestellte Konzept arbeitet in vielen Bereichen anders als es üblich ist. Das macht es schwierig, dass Konzept potentiellen Nutzern zu vermitteln. Auch ist die Entwicklung innerhalb des Konzepts aufgrund dessen aufwendig. Dies wird durch das vorgestellte Regelwerk des Konzepts verstärkt. Z.B. führen die Regeln, dass alle Methoden öffentlich und in Interfaces hinterlegt sein müssen dazu, dass für alles eine allgemeingültige Schnittstelle definiert werden muss. Der Aufwand ein Interface zu erstellen, ist dabei wesentlich höher als die Erstellung einer privaten Methode.
Außerdem wird die Reduzierung der Methoden durch die zusätzlichen Konstruktoren erkauft. In Java ist dieser Umstand kritisch, da keine Default-Werte für Argumente gesetzt werden können. Daraus folgt, dass für die Anwendung des Konzepts eine andere Programmiersprache gewählt werden sollte. Innerhalb der JVM-Sprachen bietet sich hierfür z.B. Kotlin an.

Zwar zeigte sich in der Arbeit die Umsetzbarkeit des Konzepts in den vorgestellten Bereichen, ob allerdings auch alle anderen Funktionalitäten von GUI-Frameworks umgesetzt werden können und wie sich das Konzept dabei verhält, ist zu klären. Darunter fällt unter anderem die Fokussierbarkeit von Elementen und die Umsetzung der Dynamik in Layouts, Texten und anderen Bereichen.

Literaturverzeichnis

[Alf09] Alfonse. Primitive, 2009. [Online; Stand 10. Mai 2019]. URL: http://www.khronos.org/opengl/wiki/Primitive#Point_primitives.

[Blo17] J. Bloch. *Effective Java: Third Edition*. Pearson Education, 2017.

[Bug17] Y. Bugayenko. *Elegant Objects*. Number Bd. 2 in Elegant Objects. CreateSpace Independent Publishing Platform, 2017.

[Cas] Cindy Castillo. Javafx architecture. URL: https://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm.

[Fea11] M.C. Feathers. *Effektives Arbeiten mit Legacy Code: Refactoring und Testen bestehender Software*. mitp Professional. mitp, 2011.

[GJHV11] E. Gamma, R. Johnson, R. Helm, and J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Programmer's choice. Pearson Deutschland, 2011.

[glf] Glfw. [Online; Stand 2. Juli 2019]. URL: https://www.glfw.org/docs/latest/.

[GP] Sebastian Götz and Mario Pukall. On performance of delegation in java. URL: http://www.cs.cmu.edu/~tdumitra/hotswup09/papers/hotswup09_submission_5.pdf.

[jav] Javafx. [Online; Stand 2. Juli 2019]. URL: https://github.com/javafxports/openjdk-jfx/tree/develop/modules/javafx.graphics/src/main/java/com/sun/prism.

[JK18] Michael Satran John Kennedy. Retained mode versus immediate mode, 2018. [Online; Stand 13. Mai 2019]. URL: https://docs.microsoft.com/en-us/windows/desktop/learnwin32/retained-mode-versus-immediate-mode.

[lwj] Lwjgl. [Online; Stand 2. Juli 2019]. URL: https://www.lwjgl.org/.

[Mar08] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, 2008.

[Mil56] George Armitage Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review, 63(2)*, pages 81–97, 1956.

[oraa] Graphics. [Online; Stand 22. Juni 2019]. URL: https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html.

[orab] Graphics2d. [Online; Stand 22. Juni 2019]. URL: https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html.

[orac] Jcomponent. [Online; Stand 22. Juni 2019]. URL: https://docs.oracle.com/javase/7/docs/api/javax/swing/JComponent.html.

[orad] Painting in awt and swing. [Online; Stand 3. Juli 2019]. URL: https://www.oracle.com/technetwork/java/painting-140037.html.

[orae] Vbox. [Online; Stand 2. Juli 2019]. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/VBox.html.

[PL16] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, June 2016.

[Ser19] Kapralov Sergey. Primitive, 2019. [Online; Stand 6. Juni 2019]. URL: https://www.pragmaticobjects.com/chapters/005_implementation_inheritance_paranoia.html.

[Top10] K. Topley. *JavaFX Developer's Guide*. Developer's Library. Pearson Education, 2010.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Untersuchung eines alternativen Modularisierungskonzepts für GUI-Frameworks

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original