



Fortgeschrittene Objektorientierte Entwurfstechniken

Klausur

16. Februar 2004, 8:00 Uhr bis 10:00 Uhr

Hilfsmittel

Schreibmaterial (Kugelschreiber)

Hinweise

- Verwenden Sie eigenes Papier nur als Schmierpapier!
- Verwenden Sie keine rot oder grün schreibenden Stifte und keine Bleistifte!
- Schreiben Sie Ihre Lösungen nur auf diese Klausurbögen!
- Tragen Sie Ihre Matrikelnummer auf jedem Blatt ein!
- Lösen Sie die Heftung der Klausur *nicht*!
- Bögen ohne Namen werden nicht bewertet!

Nachname	
Vorname	
Matrikelnummer	
Fachbereich	

Aufgabe	1	2	3	4	5	Σ
Punkte						
Maximum	23	17	13	22	12	87

1 Modularität und Entwurfsmuster (23 Punkte)

1.1 Kriterien und Regeln (4 Punkte)

Charakterisieren Sie *eine beliebige* der fünf Modularitätsregeln nach Meyer. Geben Sie dabei auf jeden Fall an, aus welchen der Modularitätskriterien sich diese Regel herleiten lässt. Begründen Sie Ihre Herleitung und beschreiben Sie kurz das jeweilige Kriterium bzw. die jeweiligen Kriterien.

1.2 Uniform Access Principle (4 Punkte)

Was versteht man unter dem *Uniform Access Principle*? Geben Sie je ein Beispiel für einen Verstoß dagegen und ein Einhalten des Prinzips an.

1.3 Kopplung und Kohäsion (4 Punkte)

Was versteht man unter den Begriffen Kopplung (*coupling*) und Kohäsion (*cohesion*)? Damit ein System als „modular“ bezeichnet werden kann, sollen „low coupling“ und „high cohesion“ vorliegen. Warum? Argumentieren Sie mit den Modularitätskriterien, -regeln und -prinzipien.

1.4 Single Choice Principle (4 Punkte)

Charakterisieren Sie das *Single Choice Principle*. Geben Sie anschließend ein Entwurfsmuster an, das dieses Prinzip im besonderen Maße unterstützt. Begründen Sie Ihre Antwort.

1.5 Decorator (7 Punkte)

Beschreiben Sie das *Decorator*-Entwurfsmuster. Beantworten Sie dazu folgende Fragen:

- Welches Problem soll das Muster lösen?
- Wie ist das Muster aufgebaut? Geben Sie ein Klassendiagramm an.
- Was sind die entscheidenden Vorteile des Musters gegenüber der Benutzung von Subklassen?
- Charakterisieren Sie kurz das *late binding of this*-Problem und erläutern Sie, wie es beim Einsatz des Decorator-Musters auftritt.
- Welches der Modularitätskriterien nach Meyer wird von diesem Muster in besonderem Maße unterstützt? Begründen Sie Ihre Antwort.

2 Aspektorientierte Programmierung (17 Punkte)

Betrachten Sie folgenden Code, der im Folgenden **V1** genannt wird:

```
interface FigureElement {
    void moveBy(int x, int y);
}

class Line implements FigureElement {
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) {
        this.p1 = p1;
        Display.update();
    }
    void setP2(Point p2) {
        this.p2 = p2;
        Display.update();
    }
    void moveBy(int x, int y) {
        p1.moveBy(x,y);
        p2.moveBy(x,y);
        Display.update();
    }
}

class Point implements FigureElement {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) {
        this.x = x;
        Display.update();
    }
    void setY(int y) {
        this.y = y;
        Display.update();
    }
    void moveBy(int x, int y) {
        this.x += x;
        this.y += y;
        Display.update();
    }
}

class Display {
    public static void update() {...}
    public static void show(FigureElement e) {...}
    ...
}
```

In einer zweiten Version **V2** dieses Codes werden die Aufrufe an `Display.update()` gelöscht und durch einen AspectJ-Aspekt ersetzt, der wie folgt aussieht:

```
aspect DisplayUpdating {
    pointcut move():
        call(void FigureElement.moveBy(int, int) ||
            call(void Line.setP1(Point))           ||
            call(void Line.setP2(Point))           ||
            call(void Point.setX(int))             ||
            call(void Point.setY(int)));

    after() returning: move() {
        Display.update();
    }
}
```

In einer dritten Version **V3** sieht der selbe Aspekt wie folgt aus:

```
aspect DisplayUpdating {
    pointcut move():
        call(void FigureElement.moveBy(int, int)
            || call(void FigureElement+.set*(..));

    after() returning: move() {
        Display.update();
    }
}
```

2.1 Stetigkeit von V1-V3 (9 Punkte)

Vergleichen Sie, wie stetig V1, V2 und V3 gegenüber potenziellen Änderungen an der Anwendung sind. Mit „Stetigkeit“ (Continuity) sind Kriterien gemeint wie:

- Wieviel Code muss geändert werden?
- Über wieviele verschiedene Stellen und Klassen sind die Änderungen verteilt?

Berücksichtigen Sie dabei folgende Änderungsszenarien:

- a) Das Update des Displays soll komplett ausgeschaltet werden.
- b) Die `Display.refresh()`-Methode soll mit einem zusätzlichen Parameter für das geänderte `FigureElement` ausgestattet werden.
- c) Eine neue `FigureElement`-Subklasse `Circle` wird hinzugefügt.

2.2 Weitere Szenarien (8 Punkte)

Geben Sie zwei weitere Änderungsszenarien an, welche die Stärken und Schwächen von V1-V3 illustrieren. Vergleichen Sie auch bei Ihren eigenen Änderungsszenarien V1-V3 bezüglich ihrer Stetigkeit. Ihre Änderungsszenarien sollen keine Varianten der Szenarien oben sein.

3 Architekturmuster (13 Punkte)

3.1 Layer (7 Punkte)

Eine zentrale Invariante bei einer Layer-Architektur ist, dass tiefere Layer nicht an darüberliegende Layer gekoppelt sind.

1. Beschreiben Sie kurz, wieso das *Observer*-Entwurfsmuster helfen kann, diese Invariante beizubehalten. Die Beschreibung des Observer-Patterns selber steht hierbei nicht im Vordergrund.
2. Welche Konsequenzen ergeben sich, wenn Layer $N+1$ nicht nur direkt mit Layer N sondern auch mit Layer $N-1, \dots, 1$ kommuniziert?

3.2 Pipes und Filter (6 Punkte)

Eine der zentralen Eigenschaften einer *Pipes-and-Filters*-Architektur ist ihr Potenzial für Nebenläufigkeit.

1. Diskutieren Sie die Vor- und Nachteile einer nebenläufigen Ausführung von Filtern.
2. Was sind die Konsequenzen bzgl. nebenläufiger Ausführung, wenn ein Filter verwendet wird, der die Eingabe (z. B. Textzeilen) in sortierter Reihenfolge wieder ausgibt?

4 Refactoring (22 Punkte)

Solitär ist ein Kartenspiel, das von einem Spieler alleine gespielt wird. In Abbildung 1 sehen Sie den Aufbau einer Solitär-Partie. Er besteht aus mehreren Kartenstapeln mit verschiedenen Eigenschaften. Auf den vier *Zielstapeln* (rechts oben) werden Karten jeweils einer Farbe der Reihe nach abgelegt. Sind alle Zielstapel vollständig gefüllt, ist das Spiel gewonnen. Der verdeckte Stapel links oben wird als *Talon* bezeichnet. Von ihm kann jeweils eine Karte aufgedeckt und auf die *Ablage* daneben gelegt werden. Ist der Talon leer, so wird die Ablage umgedreht und als Talon verwendet. Von der Ablage kann jeweils die oberste Karte weggenommen und auf einem Zielstapel oder einem Feld abgelegt werden. Im letzten Fall müssen die Karte, auf die gelegt wird, und die Karte von der Ablage eine Reihe bilden, d. h., der Wert der abgelegten Karte ist um eins geringer als der Wert der Karte, auf die gelegt wird (das ist eine Vereinfachung gegenüber echtem Solitär). In der unteren Reihe sind sieben Spalten. In jeder Spalte liegen ein offener Stapel, *Feld* genannt, und ein verdeckter Stapel, *Feldtalon* genannt. Von einem Feldtalon darf eine Karte aufgedeckt werden, wenn er nicht leer ist, aber das Feld leer ist. Weitere Regeln sollen nicht beachtet werden.

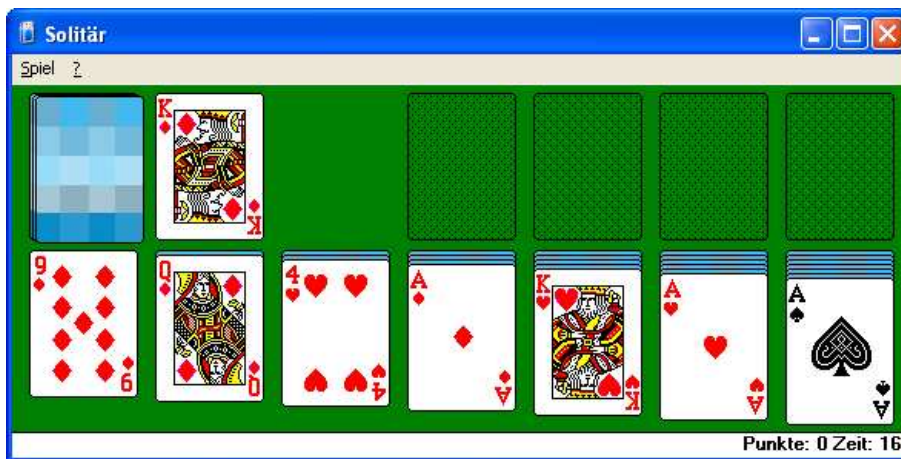


Abbildung 1: Screenshot von Solitär.

In Listing 1 sehen Sie eine Klasse, die Methoden zum Aufbauen und Lösen einer Solitär-Partie definiert. Die Karten, die hierzu verwaltet werden müssen, werden als String repräsentiert, wobei das erste Zeichen für die Farbe der Karte steht (Karo, Herz, Pik, Kreuz) und die restlichen Zeichen für den Wert (1: Ass, 2 - 10, 11: Bube, 12: Dame, 13: König).

4.1 Erkennen von Refactorings (10 Punkte)

Auf diese Klasse sollen nun Refactorings angewendet werden. Es wird zunächst ein Refactoring angewendet, dessen Ergebnis in Listing 2 zu sehen ist. In Listing 1 sind Codefragmente markiert und durchnummeriert (durch `<<<CODE X>>>` und `<<</CODE X>>>` gekennzeichnet). Das sind die Stellen, die durch das Refactoring verschoben werden. In Listing 2 sind dieselben Stellen auch markiert, so dass Sie nachvollziehen können, wohin sie verschoben wurden. Durch `<<<NEU>>>` und `<<</NEU>>>` sind hier Codefragmente markiert, die durch das Refactoring neu hinzugekommen sind.

Beschreiben Sie das durchgeführte Refactoring:

1. Fassen Sie den Mechanismus des Refactoring in maximal einem Satz zusammen (ähnlich wie die Namen der aus Vorlesung und Übung bekannten Refactorings).
2. Welches Refactoring wird erst nach Anwenden dieses Refactorings möglich? Warum war es vorher nicht anwendbar?

3. Beschreiben Sie stichwortartig das Vorgehen, um das Refactoring durchzuführen.

4.2 Durchführen von Refactorings (12 Punkte)

Gehen Sie von Listing **2** aus und beschreiben Sie zwei verschiedene Refactorings, die auf diesen Code angewendet werden können und dessen Qualität verbessern. Zählen Sie keine Refactorings auf, die nur Umbenennungen durchführen. Außerdem beachten Sie, dass das Programm vorher und nachher semantisch äquivalent sein muss. Schreiben Sie für jedes Refactoring, das Sie anwenden wollen, folgende Punkte auf:

- Name des Refactorings, bzw. kurze Charakterisierung, was es macht (maximal ein Satz).
- Welche Veränderungen werden an welchen Zeilen im Code vorgenommen?
- Was ist die Motivation für das Refactoring, bzw. woran erkennen Sie, dass es angewendet werden sollte?
- Was ist hinterher besser?

Schreiben Sie **nicht** auf, wie der Code hinterher aussieht. Für die Bearbeitung dieser Aufgabe muss **kein Quellcode geschrieben werden**. Listen Sie außerdem **nur Refactorings auf, die direkt auf den vorgestellten Code angewendet werden können**, also keine Refactorings, die erst möglich sind, nachdem ein anderes Refactoring durchgeführt wurde.

Listing 1: Klasse mit Methoden zum Aufbau und Lösen einer Solitär-Partie.

```

01 public class Solitair {
02
03     Stack[] feldTalons, felder, zielStapel;
04     Stack talon, ablage;
05
06     public void kartenGeben() { /* ... */ }
07     public void ausgeben() { /* ... */ }
08
09     public boolean kannTalonAufdecken(int feld) {
10         if(feld == -1) // Der Talon ist gemeint
11             return !talon.isEmpty() || !ablage.isEmpty();
12         else // Ein Feld Talon ist gemeint
13             return felder[feld].isEmpty() && !feldTalons[feld].isEmpty();
14     }
15
16     public void <<<CODE 1>>>loese<<</CODE 1>>>() {
17         <<<CODE 2>>>boolean fertig = false;
18
19         boolean kannFeldTalonAufdecken = false;
20         int feldTalonZumAufdecken = 0;
21
22         boolean kannTalonAufdecken = false;
23
24         boolean kannKartenBewegen = false;
25         int kartenBewegenVonFeld = 0, kartenBewegenNachFeld = 0, anzahlKartenBewegen = 0;<<</CODE 2>>>
26
27         <<<CODE 3>>>while(!fertig) {
28             kannFeldTalonAufdecken = false;
29             kannKartenBewegen = false;
30             kannTalonAufdecken = false;
31
32             // findet den Feld Talon am weitesten rechts, der aufgedeckt werden kann
33             for(int i = 0; i < 7; i++) {
34                 if(kannTalonAufdecken(i)) {
35                     kannFeldTalonAufdecken = true; feldTalonZumAufdecken = i;
36                 } }
37
38             // kann der Talon aufgedeckt werden?
39             if(kannTalonAufdecken(-1))
40                 kannTalonAufdecken = true;
41
42             // kann die oberste Karte von der Ablage auf dem Feld angelegt werden?
43             if(!ablage.isEmpty()) {
44                 String karte = (String) ablage.peek();
45                 String wertString = karte.substring(1);
46                 int ablageWert = Integer.parseInt(wertString);
47
48                 for(int i = 0; i < 7; i++) {
49                     karte = (String) ablage.peek();
50                     wertString = karte.substring(1);
51                     int wert = Integer.parseInt(wertString);
52
53                     if(ablageWert == wert + 1) {
54                         kannKartenBewegen = true;
55                         kartenBewegenVonFeld = -1; kartenBewegenNachFeld = i; anzahlKartenBewegen = 1;
56                     } } }
57
58             if(kannFeldTalonAufdecken) { /* ... (Talon aufdecken) */ }
59             else if(kannKartenBewegen) { /* ... (Karten bewegen) */ }
60             else if(kannTalonAufdecken) { /* ... (Talon aufdecken) */ }
61             else // keine Züge waren möglich: Abbruch
62                 fertig = true;
63         } }<<</CODE 3>>> }

```

Listing 2: Ergebnis der Anwendung eines Refactorings auf die Klasse in Listing 1.

```

01 public class Solitair {
02     Stack[] feldTalons, felder, zielStapel;
03     Stack talon, ablage;
04     public void kartenGeben() { /* ... */ }
05     public void ausgeben() { /* ... */ }
06     public boolean kannTalonAufdecken(int feld) {
07         if(feld == -1) // Der Talon ist gemeint
08             return !talon.isEmpty() || !ablage.isEmpty();
09         else // Ein Feld Talon ist gemeint
10             return felder[feld].isEmpty() && !feldTalons[feld].isEmpty();
11     }
12     public void loese() {
13         <<<NEU>>new Loese(this).compute();<<</NEU>>
14     } }
15
16 <<<NEU>>public class<<</NEU>> <<<CODE 1>>Loese<<</CODE 1>> <<<NEU>>{
17
18     private final RefactoredSolitair solitair;<<</NEU>>
19     <<<CODE 2>>private boolean fertig = false;
20     private boolean kannFeldTalonAufdecken = false;
21     private int feldTalonZumAufdecken = 0;
22     private boolean kannTalonAufdecken = false, kannKartenBewegen = false;
23     private int kartenBewegenVonFeld = 0, kartenBewegenNachFeld = 0, anzahlKartenBewegen = 0;<<</CODE 2>>
24
25     <<<NEU>>public Loese(RefactoredSolitair solitair) {
26         this.solitair = solitair;
27     }
28
29     public void compute() {<<</NEU>>
30         <<<CODE 3>>while(!fertig) {
31             kannFeldTalonAufdecken = false;
32             kannKartenBewegen = false;
33             kannTalonAufdecken = false;
34
35             // findet den Feld Talon am weitesten rechts, der aufgedeckt werden kann
36             for(int i = 0; i < 7; i++) {
37                 if(kannTalonAufdecken(i)) {
38                     kannFeldTalonAufdecken = true; feldTalonZumAufdecken = i;
39                 } }
40
41             // kann der Talon aufgedeckt werden?
42             if(kannTalonAufdecken(-1))
43                 kannTalonAufdecken = true;
44
45             // kann die oberste Karte von der Ablage auf dem Feld angelegt werden?
46             if(!ablage.isEmpty()) {
47                 String karte = (String) ablage.peek();
48                 String wertString = karte.substring(1);
49                 int ablageWert = Integer.parseInt(wertString);
50
51                 for(int i = 0; i < 7; i++) {
52                     karte = (String) ablage.peek();
53                     wertString = karte.substring(1);
54                     int wert = Integer.parseInt(wertString);
55
56                     if(ablageWert == wert + 1) {
57                         kannKartenBewegen = true;
58                         kartenBewegenVonFeld = -1; kartenBewegenNachFeld = i; anzahlKartenBewegen = 1;
59                     } } }
60
61             if(kannFeldTalonAufdecken) { /* ... (Talon aufdecken) */ }
62             else if(kannKartenBewegen) { /* ... (Karten bewegen) */ }
63             else if(kannTalonAufdecken) { /* ... (Talon aufdecken) */ }
64             else // keine Züge waren möglich: Abbruch
65                 fertig = true;
66 }<<</CODE 3>> <<<NEU>>} <<</NEU>>

```

5 Multiple Choice (12 Punkte)

Kreuzen Sie bei den folgenden Fragen die richtigen Kästchen an. Für jedes richtig gesetzte Kreuz wird ein Punkt vergeben, jedes falsch gesetzte Kreuz führt zum Abzug eines Punktes. In dieser Aufgabe sind minimal null Punkte zu erlangen. Bei einigen Fragen sind mehrere Antworten richtig.

Ein Entwurfsmuster wird...

- gefunden.
- erfunden.
- generiert.

Was sind mögliche Vorteile, wenn ein Pattern von einer Sprache direkt unterstützt wird?

- Das Pattern kann erzwungen werden, man muss sich nicht auf Konventionen verlassen.
- Der Code ist leichter verständlich, weil er auf einer höheren Abstraktionsebene ist.
- Das Pattern kann dadurch nur noch an Stellen angewendet werden, an denen es Sinn ergibt.

Design Patterns sind abstrakter als Frameworks.

- Richtig.
- Falsch.

White-Box-Frameworks werden konfiguriert durch...

- Komposition
- Vererbung
- Komposition und Vererbung

Bei Frameworks gibt es zwei Arten von Benutzern: den Benutzer A, der es konfiguriert und den Benutzer B, der die konfigurierte Frameworkinstanz einsetzt. Bei einem Black-Box-Framework...

- sieht Benutzer A eine detailliertere Schnittstelle des Frameworks als Benutzer B.
- sieht Benutzer B eine detailliertere Schnittstelle des Frameworks als Benutzer A.
- sehen beide Benutzer die gleiche Schnittstelle des Frameworks.

Auf welcher Ebene der OMG-Standardarchitektur für Metaebenen sind UML-Modelle angesiedelt?

- M1
- M2
- M3
- M4

Datenstrukturen wie Listen, Arrays, Stacks etc. bilden eine...

- horizontale Domäne.
- vertikale Domäne.

Model-Driven Architecture (MDA)...

- ist ein Ansatz zur generativen Programmierung.
- geht über generative Programmierung hinaus.

Was ist der Unterschied zwischen der asymmetrischen und der symmetrischen Konfliktbeseitigung bei Multiple Dispatch?

- Bei der asymmetrischen Lösung wird zur Compile-Zeit eine Fehlermeldung generiert; bei der symmetrischen Lösung ein Laufzeitfehler.
- Bei der asymmetrischen Lösung wird künstlich eine Eindeutigkeit hergestellt, indem man über die Reihenfolge der Argumente Prioritäten vergibt; bei der symmetrischen Lösung werden Konflikte konservativ erkannt und ggf. eine Fehlermeldung erzeugt.
- Bei der symmetrischen Lösung werden Methoden im Unterschied zur asymmetrischen Lösung nicht mehr via `receiver.meth(arg1,arg2)` sondern via `meth(receiver, arg1, arg2)` aufgerufen und so die Konflikte beseitigt.

Sie müssen sich in ihrem Java-Projekt zwischen einer funktionalen und einer objekt-orientierten Dekomposition entscheiden.

- In Java kann man nur objekt-orientiert zerlegen, weil Java eine OO-Sprache ist.
- Die funktionale Dekomposition ist vorzuziehen.
- Die objekt-orientierte Dekomposition ist vorzuziehen.
- Die Antwort hängt davon ab, was für Arten von Erweiterungen der Software zu erwarten sind.