

Design Pattern in Java-Programmen

Gesamtinhaltsverzeichnis

1	Einleitung.....	1-3
1.1	Zum Aufbau und zur Benutzung des Skripts	1-3
1.2	Aufbau des Workspaces.....	1-5
1.3	Utility-Klassen.....	1-6
1.3.1	Die Db-Utilities	1-6
1.3.2	Die Utlity-Klasse XMLScanner	1-7
1.4	Einige Gamma-Zitate.....	1-8
2	Erzeugungsmuster.....	2-3
2.1	Abstract Factory	2-4
2.1.1	Problem	2-5
2.1.2	Lösung	2-9
2.1.3	Bean-Implementierungen.....	2-12
2.1.4	JDBC	2-16
2.2	Builder	2-22
2.2.1	Problem	2-23
2.2.2	Lösung	2-28
2.2.3	Ein anderer Builder	2-31
2.3	Factory Method.....	2-34
2.3.1	Problem	2-34
2.3.2	Lösung	2-37
2.3.3	Delegation via Interface	2-39
2.4	Prototype	2-42
2.4.1	Problem	2-43
2.4.2	Lösung	2-45
2.5	Singleton	2-47
2.5.1	Problem	2-48
2.5.2	Lösung	2-49
2.5.3	Lazy Creation.....	2-51
2.5.4	Serialisierungs-Problem.....	2-52
2.5.5	ReadResolve	2-53
2.5.6	Singletons als einfache enums	2-55
2.5.7	Tripleton.....	2-56
2.5.8	Chess.....	2-61
2.5.9	Runtime und Toolkit.....	2-62

3	Strukturmuster	3-3
3.1	Adapter	3-4
3.1.1	Problem	3-5
3.1.2	Lösung	3-7
3.1.3	Klassen-Adapter	3-9
3.1.4	Listener-Registratur	3-11
3.1.5	Runnable	3-14
3.1.6	Lambdas	3-17
3.1.7	Ad-hoc-Lambdas	3-18
3.1.8	Swing: ActionListener	3-19
3.1.9	Swing: TreeModel	3-23
3.2	Bridge	3-27
3.2.1	Problem	3-28
3.2.2	Lösung	3-33
3.2.3	Figures und Drawers	3-36
3.3	Decorator	3-39
3.3.1	Problem	3-40
3.3.2	Lösung	3-44
3.3.3	Reader	3-46
3.4	Facade	3-50
3.4.1	Problem	3-51
3.4.2	Lösung	3-53
3.5	Flyweight	3-56
3.5.1	Problem	3-57
3.5.2	Lösung	3-60
3.5.3	Ein String-Cache	3-63
3.6	Composite	3-65
3.6.1	Problem	3-66
3.6.2	Lösung	3-70
3.6.3	Expressions	3-72
3.7	Proxy	3-77
3.7.1	Problem	3-78
3.7.2	Lösung	3-79
3.7.3	InvocationHandler	3-82
3.7.4	DynamicProxy	3-86
3.7.5	SimpleInvocationHandler	3-88
4	Verhaltensmuster	4-5
4.1	Command	4-6
4.1.1	Problem	4-7

4.1.2	Lösung	4-9
4.1.3	History	4-13
4.1.4	CommadFactory	4-18
4.2	Observer	4-20
4.2.1	Problem	4-21
4.2.2	Lösung	4-23
4.2.3	Lambdas	4-26
4.2.4	Vereinfachung	4-28
4.2.5	Observer/Observable	4-29
4.3	Visitor	4-32
4.3.1	Problem	4-33
4.3.2	Lösung	4-37
4.3.3	Dispatch mit Reflection	4-41
4.3.4	Dispatch mit Consumer	4-44
4.3.5	Lambdas	4-47
4.3.6	Swing-Beispiel	4-48
4.4	Interpreter	4-51
4.4.1	Problem	4-52
4.4.2	Lösung	4-54
4.4.3	Bau einer Datenstruktur	4-56
4.4.4	BeanFactory-Beispiel	4-59
4.5	Iterator	4-66
4.5.1	Problem	4-67
4.5.2	Lösung	4-70
4.5.3	Das Interface Iterable	4-74
4.5.4	Interner Iterator	4-76
4.5.5	Tree-Iterator	4-79
4.6	Memento	4-81
4.6.1	Problem	4-82
4.6.2	Lösung	4-85
4.6.3	Serialisierung	4-88
4.6.4	Ein grafischer Editor	4-90
4.7	Template Method	4-91
4.7.1	Problem	4-92
4.7.2	Lösung	4-94
4.7.3	Delegation via Interface	4-98
4.7.4	Erweiterung der Delegation	4-101
4.7.5	Ein Gruppenwechsel-Prozessor	4-103
4.8	Strategy	4-107
4.8.1	Problem	4-108

4.8.2	Lösung	4-110
4.8.3	Lambdas	4-112
4.8.4	XYLayout	4-113
4.8.5	Functions	4-115
4.8.6	Circuits	4-120
4.9	Mediator	4-126
4.9.1	Problem	4-127
4.9.2	Lösung	4-129
4.9.3	EventBus	4-131
4.9.4	Queue	4-137
4.10	State	4-142
4.10.1	Problem	4-143
4.10.2	Lösung	4-145
4.10.3	Stack	4-148
4.10.4	FigureEditor	4-151
4.10.5	Dispatcher	4-153
4.11	Chain of Responsibility	4-158
4.11.1	Problem	4-159
4.11.2	Lösung	4-160
4.11.3	JDBC-Driver und DriverManager	4-164
5	Literaturverzeichnis	5-3

1

Einleitung

1.1	Zum Aufbau und zur Benutzung des Skripts	1-3
1.2	Aufbau des Workspaces.....	1-5
1.3	Utility-Klassen.....	1-6
1.3.1	Die Db-Utilities	1-6
1.3.2	Die Uitlity-Klasse XMLScanner	1-7
1.4	Einige Gamma-Zitate.....	1-8

1 Einleitung

1.1 Zum Aufbau und zur Benutzung des Skripts

Das Skript demonstriert die bei Gamma et. al. beschriebenen Entwurfsmuster. Entwurfsmuster, die darüber hinausgehen (Enterprise-Pattern, Threading-Pattern etc.), können hier nicht berücksichtigt werden.

Jedes Entwurfsmuster wird eingeleitet mit einer allgemeinen Zweckbestimmung (wobei die Formulierung jeweils von Gamma wörtlich übernommen wird – genauer als Gamma kann man die Sache ohnehin nicht ausdrücken). Dann folgen jeweils einige Beispiele, anhand derer der Sinn der Muster verdeutlicht werden soll.

Grundsätzlich existieren zu jedem dieser Beispiele ein Klassendiagramm und der komplette Java-Code. Zusätzlich sind am Ende eines jeden Abschnitts einige Fragen und Aufgaben beschrieben, anhand derer das Thema vertieft werden kann.

Gamma beschreibt bei jedem Muster die Teilnehmer, die Interaktionen, die Vorteile und die Einschränkungen. Dieses Skript verzichtet darauf, diese Beschreibungen zu wiederholen. Das Skript soll und kann die Lektüre des Originals nicht ersetzen.

Der gesamte Quellcode basiert auf der Java 8. Die neuen Features von Java 8 sind konsequent übernommen worden (insbesondere die Lambdas).

Die Gliederung von Gamma ist 1:1 übernommen worden (Erzeugungsmuster, Strukturmuster und Verhaltensmuster – und deren Unterkapitel). Hierin kann der Vorteil gesehen werden, dass das Skript nahe am "Original" bleibt. Was das Studium der Beispiele angeht, ist diese Reihenfolge aber die denkbar schlechteste. U.a. deshalb, weil etwa zur Demonstration einiger Erzeugungsmuster bereits andere Muster (Verhaltensmuster) vorausgesetzt werden (so wird bereits im ersten Kapitel zum Thema Factory Method das Entwurfsmuster Template Method vorausgesetzt). Deshalb hier einige Hinweise, in welcher Reihenfolge die Beispiele studiert werden können.

Es erscheint sinnvoll, zunächst mit Beispielen zum Thema Verhaltensmuster anzufangen. Hier bietet es sich etwa an, mit dem Template-Method Pattern zu beginnen – und etwa mit dem Strategy-, dem State- und dem Observer-Pattern fortzufahren. Diese Muster sind allesamt unmittelbar "eingängig" (eingängiger jedenfalls als die Erzeugungsmuster).

Anschließend können wichtige Strukturmuster studiert werden. Die wichtigsten dieser Muster sind wohl das Adapter-, das Composite- und das Proxy-Muster.

Die Erzeugungsmuster können ganz am Ende studiert werden. Hierbei sollte man sich allerdings immer auch vor Augen halten, dass Java mit seinem Reflection-Konzept zusätzliche Möglichkeiten der dynamischen Erzeugung von Objekten bietet, die bei Gamma nicht vorgesehen sind.

Die Zitate aus dem Werk von Gamma et. al. beziehen sich auf die 1. Auflage, 1996, Addison Wesley.

Ein letzter aber wichtiger Hinweis:

Nur eine einzige Anwendung, die in diesem Skript beschrieben ist, nutzt Multithreading. Multithreading ist nicht das Thema dieses Skripts. Das resultiert darin, dass einige der Anwendungen, die hier vorgestellt werden, in einem Multithreading-Kontext nicht sicher sind. In einem solchen Kontext müssten sicherlich manche Zugriffe synchronisiert werden; oder es müssten CopyOnWrite-Mechanismen genutzt werden etc.

1.2 Aufbau des Workspaces

Der Workspace enthält zwei Projekte, die Utility-Klassen enthalten, die in einer Vielzahl der weiteren Projekte verwendet werden: `common-util` und `db-util`.

Im `dependencies`-Projekt liegt die HSQLDB und die Derby-Datenbank.

Alle anderen Projekte sind mit `x` präfixiert und haben eine sechsstellige Nummer. Diese ist wie folgt zu interpretieren:

Die ersten beiden Stellen sind der "Kapitel"-Name. Es gibt die Kapitel 02, 03 und 04. Die nächsten beiden Stellen sind der "Abschnitts"-Name; die letzten beiden Stellen sind der "Unterabschnitts"-Name. Dann folgend die Namen des Kapitels, des Abschnitts und des Unterabschnitts. Diese Kapitel-Abschnitts-Unterabschnitts-Struktur ist exakt dieselbe, die auch diesem Skript zugrunde liegt. Es fällt somit leicht, von Skript sofort zu dem entsprechenden Projekt zu wechseln bzw. umgekehrt vom Projekt zu der entsprechenden Stelle im Skript zu wechseln.

Der erste Abschnitt jedes Projekts präsentiert zunächst ein Problem (und heißt deshalb auch so). Der zweite Abschnitt zeigt dann die Pattern-basiert Lösung dieses Problems (und heißt daher auch jeweils "Solution"). Dann folgen i.d.R. einige weitere Beispiele, welche die Bedeutung des jeweiligen Patterns näher erläutern.

1.3 Utility-Klassen

Einige der Projekte, die in diesem Skript beschrieben sind, benutzen die Datenbank; einige andere befassen benutzen XML-Dateien als Eingabedateien. Um mit der Datenbank und mit XML-Dateien einigermaßen bequem umgehen zu können, existieren einige Utility-Klassen.

1.3.1 Die Db-Utilities

Für das Testen von Datenbankanwendungen ist es wichtig, dass vor jedem Programmlauf die Datenbank wieder neu aufgebaut wird – damit das Programm immer wieder dieselbe Ausgangssituation vorfindet. Dies kann mit der `Db`-Klasse automatisiert werden.

Diese Klasse ist im Projekt `db-util` implementiert.

Die Verbindungs-Parameter für die Datenbank entnimmt dieses Tool der Datei `db.properties` (die ebenfalls im Projekt `db-util` hinterlegt ist).

Das Tool setzt weiterhin voraus, dass in dem jeweiligen Projekt eine Datei `create.sql` existiert. Diese enthält die `CREATE-SQL`-Anweisungen zum Aufbau der Datenbank (und evtl. bereits auch `INSERT-SQL`-Anweisungen).

Diese `create.sql`-Datei kann automatisch ausgeführt werden. Hierzu wird am Anfang der `main`-Methode die Methode `Db.aroundAppl()` aufgerufen. Diese Methode löscht alle Tabellen der Datenbank und führt dann die `create.sql` aus. Sie richtet zudem einen Shutdown-Hook ein, der am Ende des Programms die Inhalte aller Tabellen auf der Konsole ausgibt.

(Das Tool setzt voraus, dass sowohl `db.properties` als auch `create.sql` über den `CLASSPATH` auffindbar sind.)

1.3.2 Die Utility-Klasse XMLScanner

Zum Lesen der XML-Datei verwenden wir einen selbstgestrickten XMLScanner.

Dieser ist im Projekt `comon-utils` implementiert.

Was die genaue Implementierung betrifft, so sei auf den Quellcode verwiesen. Hier genügt die Darstellung der öffentlichen Schnittstelle dieser Klasse:

```
package util;
// ...
public class XMLScanner {

    public XMLScanner(InputStream in) throws Exception { ... }

    public boolean isStart(String name) { ... }

    public void start(String name) throws Exception { ... }

    public void end(String name) throws Exception { ... }

    public String startTextEnd(String name) throws Exception { ... }
}
```

Eine kurze Beschreibung der Methoden:

- Mittels `isStart` kann geprüft werden, ob das "aktuelle Symbol" des Scanners ein Start-Tag mit dem Namen `name` ist.
- Sofern das aktuelle Symbol des Scanners ein Start-Tag namens `name` ist, kann der Scanner mit der `start`-Methode zum nächsten Symbol weitergetrieben werden. Falls nicht, wird eine Exception geworfen.
- Sofern das aktuelle Symbol des Parsers ein Ende-Tag namens `name` ist, kann der Scanner mit der `end`-Methode zum nächsten Symbol weitergetrieben werden. Falls nicht, wird eine Exception geworfen.
- Sofern das aktuelle Symbol des Scanners ein Start-Tag namens `name` ist, kann mit der `startTextEnd`-Methode ein komplettes "inneres" Element gelesen werden: Das Start-Tag wird überlesen, der innere Text und das Ende-Tag. Der überlesene innere Text wird als Resultat zurückgeliefert.

(Natürlich hätte man darauf verzichten können, der `end`-Methode den Namen des XML-Elements zu übergeben. Indem aber der Name verlangt wird, können Programmierfehler entdeckt und diagnostiziert werden – z.B. fehlende oder zu viele `end`-Aufrufe.)

1.4 Einige Gamma-Zitate

Als Einstieg hier einige Zitate aus dem Entwurfsmuster-Buch von Gamma et al:

"Unerfahrene Entwickler ... kapitulieren oft vor der großen Anzahl der Entwurfsmöglichkeiten und greifen auf nicht objektorientierte, zuvor aber einmal von ihnen verwendete Techniken zurück. Es dauert lange, bis Anfänger verstehen, worum es bei gutem objektorientierten Entwurf eigentlich geht. Offenkundig wissen erfahrene Entwickler etwas, was unerfahrene Entwickler nicht wissen. Was ist das?

Experten wissen es zu *vermeiden*, jedes Problem von Grund auf neu anzugehen. Stattdessen verwenden sie Lösungen wieder, die sie zuvor erfolgreich eingesetzt haben. Haben sie einmal eine gute Lösung gefunden, verwenden sie diese wieder und wieder. Solche Erfahrung ist Teil dessen, was sie zu Experten macht. ...

Der Zweck dieses Buches ist es, derartige Erfahrungen beim Entwurf objektorientierter Software als Entwurfsmuster aufzuzeichnen." (S. 1f)

"Die Entwurfsmuster in diesem Buch sind *Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.*" (S. 4)

"Keines der Entwurfsmuster in diesem Buch beschreibt neuartige Entwürfe. Wir haben lediglich solche Entwürfe berücksichtigt, die mehrfach angewendet wurden und sich in unterschiedlichen Systemen bewährt haben. ... Sie gehören ... zum Allgemeinwissen objektorientierter Entwickler..." (S. 2)

2

Erzeugungsmuster

2.1	Abstract Factory	2-4
2.1.1	Problem	2-5
2.1.2	Lösung	2-9
2.1.3	Bean-Implementierungen.....	2-12
2.1.4	JDBC	2-16
2.2	Builder	2-22
2.2.1	Problem	2-23
2.2.2	Lösung	2-28
2.2.3	Ein anderer Builder	2-31
2.3	Factory Method.....	2-34
2.3.1	Problem	2-34
2.3.2	Lösung	2-37
2.3.3	Delegation via Interface	2-39
2.4	Prototype	2-42
2.4.1	Problem	2-43
2.4.2	Lösung	2-45
2.5	Singleton	2-47
2.5.1	Problem	2-48
2.5.2	Lösung	2-49
2.5.3	Lazy Creation.....	2-51
2.5.4	Serialisierungs-Problem.....	2-52

2.5.5	ReadResolve	2-53
2.5.6	Singletons als einfache enums	2-55
2.5.7	Tripletton.....	2-56
2.5.8	Chess.....	2-61
2.5.9	Runtime und Toolkit.....	2-62

2 Erzeugungsmuster

"Entwurfsmuster, die der Erzeugung von Objekten dienen, verstecken den Erzeugungsprozess. Sie helfen, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden..."

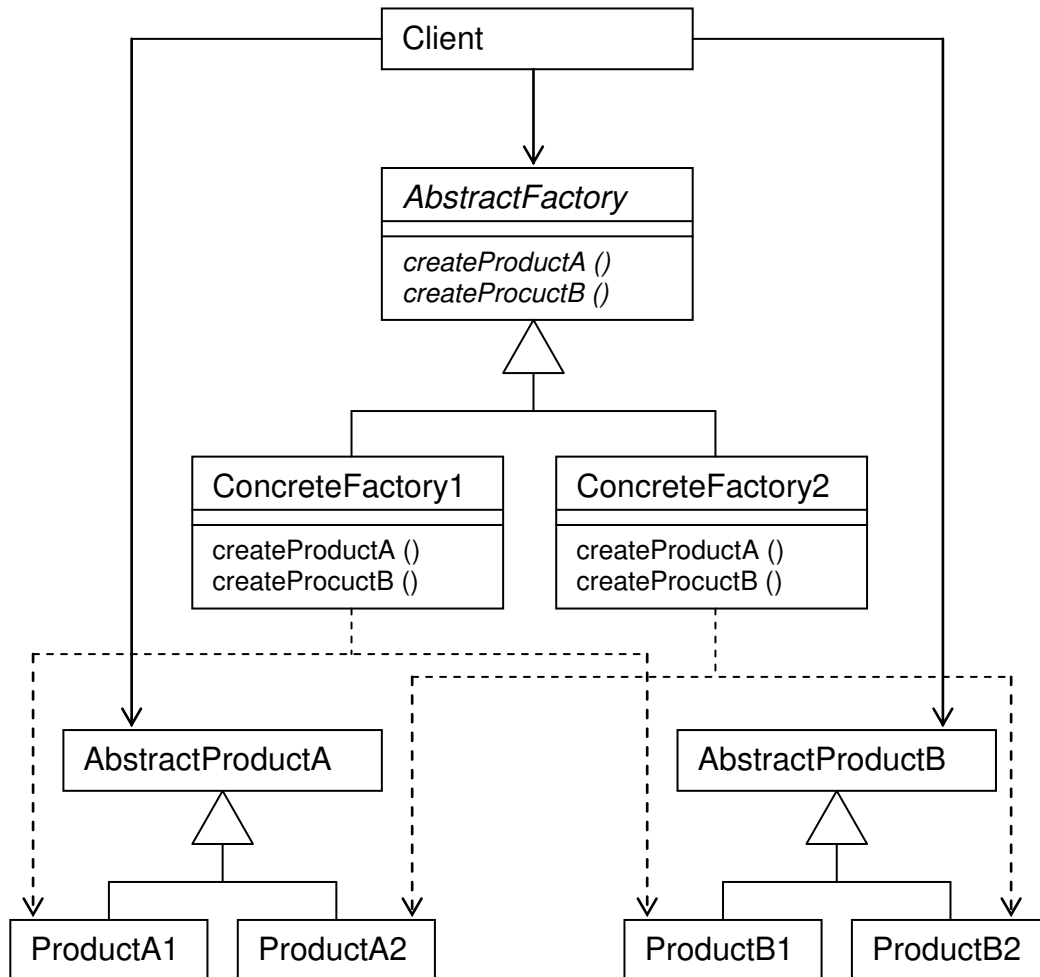
Erzeugungsmuster sind vor allem dann von Bedeutung, wenn Systeme beginnen, mehr von Objektkomposition als von Vererbung abzuhängen. Dabei bewegt sich die Konzentration von der Programmierung festgelegten Verhaltens weg. Sie bewegt sich hin zur Definition einer kleineren Menge grundlegender Verhaltenseinheiten, die zu beliebig komplexen Verhalten zusammengesetzt werden können..."

Es gibt zwei immer wiederkehrende Leitmotive in diesen Mustern. Zum einen kapseln sie alle das Wissen um die konkreten vom System verwendeten Klassen. Zum anderen verstecken sie, wie Exemplare dieser Klassen erzeugt und zusammengefügt werden..."

(Gamma, 88)

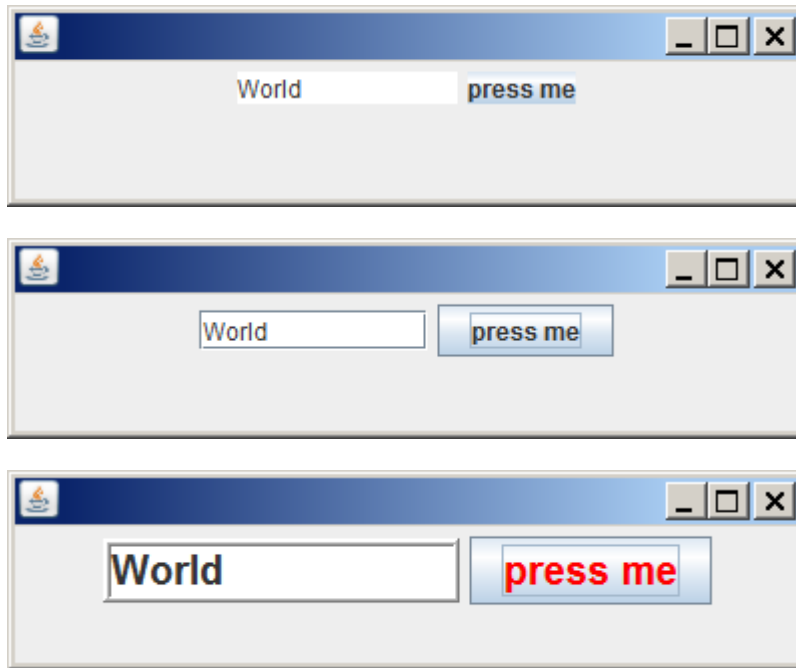
2.1 Abstract Factory

"Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen." (Gamma, 93)

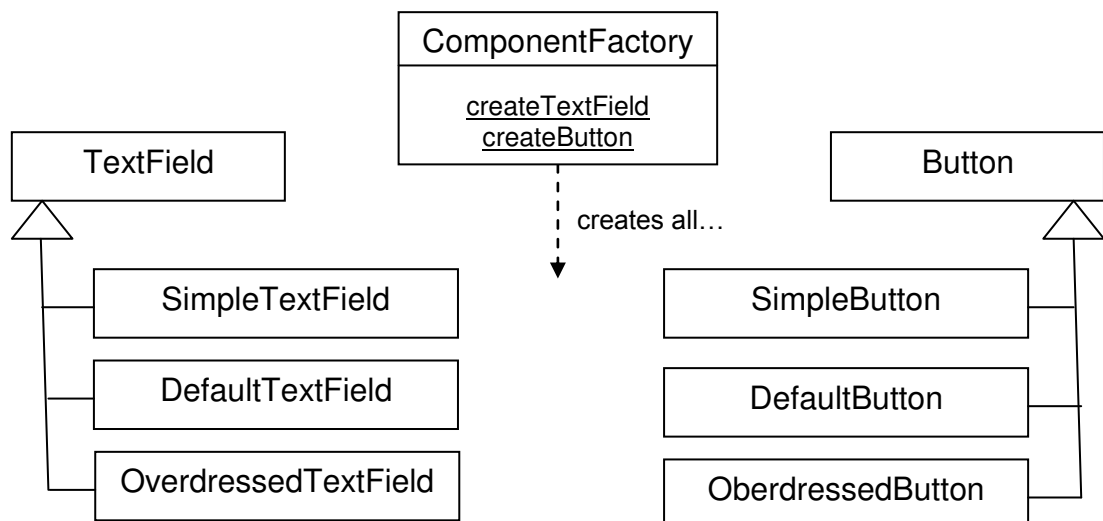


2.1.1 Problem

Die Benutzer einer GUI-Anwendung haben unterschiedliche Vorlieben. Sie möchten die Oberfläche jeweils in einer der drei folgenden Formen präsentiert bekommen (und morgen kommt ein weiterer Benutzer hinzu, der wiederum seine eigene Vorstellung von einer Oberfläche hat):



Wir könnten folgende von `TextField` und `Button` abgeleitete Klassen schreiben (und analoge Ableitungen benötigen wir natürlich auch für `JLabel`, `JList` etc.):



```
package components;
// ...
public class SimpleTextField extends JTextField {
    public SimpleTextField(int length) {
        super(length);
        this.setBorder(null);
    }
}
```

```
package components;
// ...
public class SimpleButton extends JButton {
    public SimpleButton(String text) {
        super(text);
        this.setBorder(null);
    }
}
```

```
package components;
// ...
public class DefaultTextField extends JTextField {
    public DefaultTextField(int length) {
        super(length);
    }
}
```

```
package components;
// ...
public class DefaultButton extends JButton {
    public DefaultButton(String text) {
        super(text);
    }
}
```

```
package components;
// ...
public class OverdressedTextField extends JTextField {
    private static final Font font = new Font("Arial", Font.BOLD, 20);
    public OverdressedTextField(int length) {
        super(length);
        this.setBorder(BorderFactory.createCompoundBorder(
            BorderFactory.createBevelBorder(BevelBorder.RAISED),
            BorderFactory.createBevelBorder(BevelBorder.LOWERED)));
        this.setFont(OverdressedTextField.font);
    }
}
```

```
package components;
// ...
public class OverdressedButton extends JButton {
    private static final Font font = new Font("Arial", Font.BOLD, 20);
    public OverdressedButton(String text) {
        super(text);
        this.setFont(OverdressedButton.font);
        this.setForeground(Color.red);
    }
}
```

(Es sei hier einmal dahingestellt, ob es ratsam ist, von "großen" Klassen neue Klassen abzuleiten, die nun ein Minimum an neuer oder geänderter Funktionalität anbieten. Hier mag es sinnvoller sein, einen Traversierungs-Mechanismus zu benutzen, wie er im Abschnitt zum Iterator-Pattern vorgestellt wird.)

Wir könnten dann eine einfache `ComponentFactory` schreiben (mit ausschließlich statischen Elementen):

```
package appl;
// ...
public class ComponentFactory {

    private static int type = 0;

    public static JTextField createTextField(int length) {
        switch(type) {
            case 0:
                return new SimpleTextField(length);
            case 1:
                return new DefaultTextField(length);
            case 2:
                return new OverdressedTextField(length);
            default:
                throw new RuntimeException();
        }
    }

    public static JButton createButton(String text) {
        switch(type) {
            case 0:
                return new SimpleButton(text);
            case 1:
                return new DefaultButton(text);
            case 2:
                return new OverdressedButton(text);
            default:
                throw new RuntimeException();
        }
    }
}
```

Abhängig vom Wert des `type`-Attributs werden jeweils Komponenten unterschiedlichen Typs zurückgeliefert.

Die Anwendung:

```
package appl;
// ...
public class Application extends JFrame {

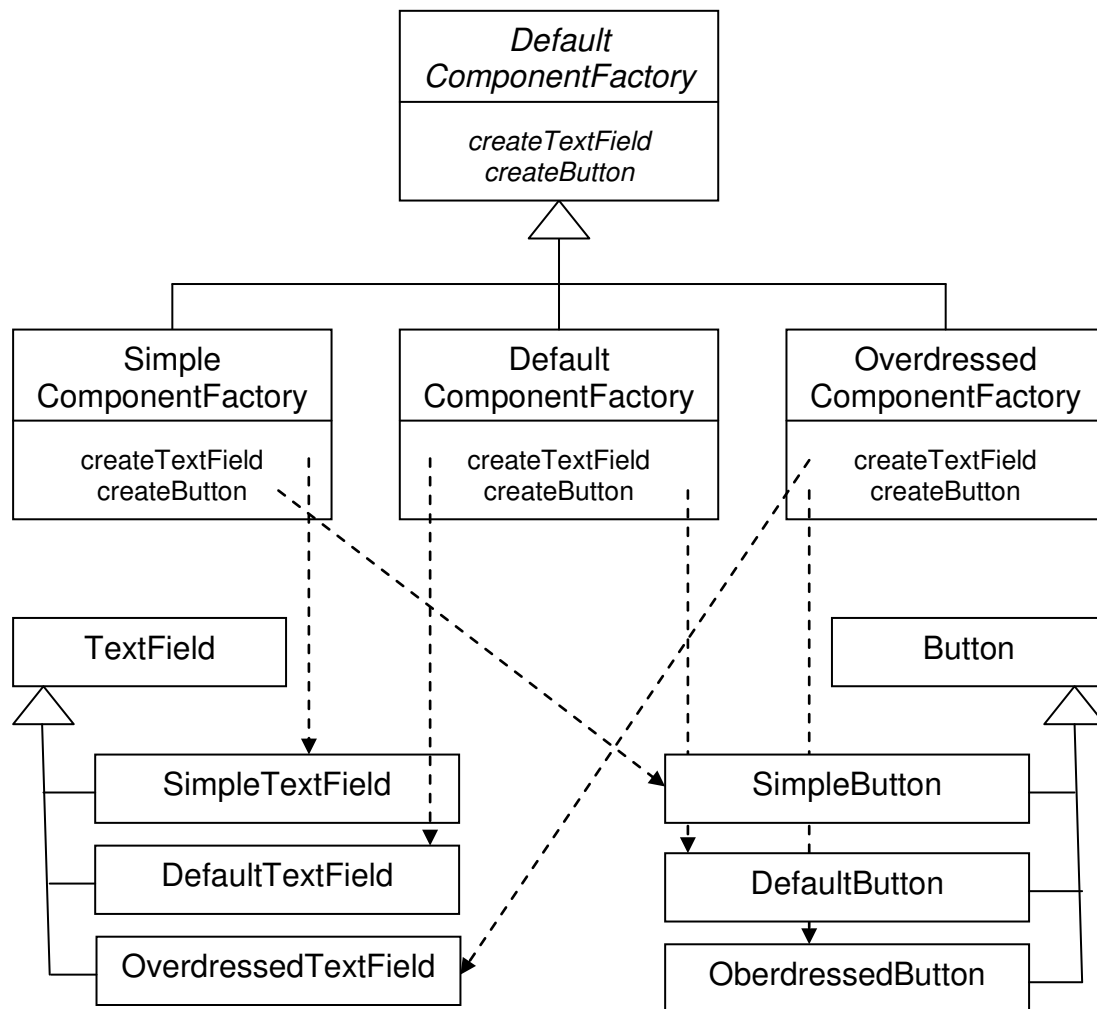
    public static void main(String[] args) {
        new Application();
    }

    private final JTextField textField =
        ComponentFactory.createTextField(10);
    private final JButton button =
        ComponentFactory.createButton("press me");

    public Application() {
        this.setLayout(new FlowLayout());
        this.add(this.textField);
        this.add(this.button);
        this.button.addActionListener(e ->
this.textField.setText("World"));
        this.setBounds(100, 100, 400, 100);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
}
```

Verlangt nun ein Benutzer ein neues Erscheinungsbild, müssen alle switch-Anweisungen um einen weiteren `case` erweitert werden. Solche switch-Anweisungen, die bei Bedarf immer wieder erweitert werden müssen, sind unschön.

2.1.2 Lösung



Wir definieren ein Interface:

```
package appl;
// ...
public interface ComponentFactory {
    public abstract JTextField createTextField(int length);
    public abstract JButton createButton(String text);
}
```

Wir implementieren das Interface in drei verschiedenen Klassen:

```
package appl;
// ...
public class SimpleComponentFactory implements ComponentFactory {
    @Override
    public JTextField createTextField(int length) {
        return new SimpleTextField(length);
    }
    @Override
    public JButton createButton(String text) {
```

```
        return new SimpleButton(text);
    }
}
```

```
package appl;
// ...
public class DefaultComponentFactory implements ComponentFactory {
    @Override
    public JTextField createTextField(int length) {
        return new DefaultTextField(length);
    }
    @Override
    public JButton createButton(String text) {
        return new DefaultButton(text);
    }
}
```

```
package appl;
// ...
public class OverdressedComponentFactory implements ComponentFactory {
    @Override
    public JTextField createTextField(int length) {
        return new OverdressedTextField(length);
    }
    @Override
    public JButton createButton(String text) {
        return new OverdressedButton(text);
    }
}
```

In der Klasse `Configuration` wird die tatsächlich benutzte `ComponentFactory` festgelegt:

```
package appl;

public class Configuration {
    private static final ComponentFactory componentFactory;
    static {
        componentFactory = new OverdressedComponentFactory();
    }
    public static ComponentFactory getComponentFactory() {
        return componentFactory;
    }
}
```

Die `ComponentFactory`, die hier instanziiert wird, könnte aufgrund eines Eintrags in einer Parameterdatei festgelegt werden. Wäre dort der Name der zu nutzenden Factory-Klasse hinterlegt, könnte via Reflection eine entsprechende Instanz dieser Klasse erzeugt werden. Man müsste den verschiedenen Benutzern dann nur jeweils eine andere Parameterdatei mitgeben...

Hier die Anwendung:

```
package appl;
// ...
public class Application extends JFrame {

    public static void main(String[] args) {
        new Application();
    }

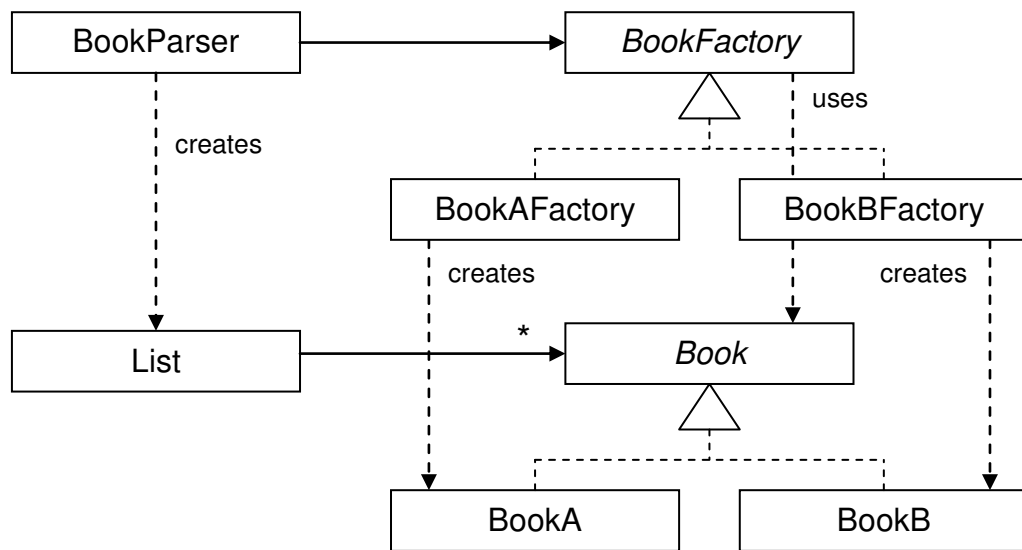
    private final ComponentFactory f =
Configuration.getComponentFactory();

    private final JTextField textField = this.f.createTextField(10);
    private final JButton button = this.f.createButton("press me");

    public Application() {
        this.setLayout(new FlowLayout());
        this.add(this.textField);
        this.add(this.button);
        this.button.addActionListener(e ->
this.textField.setText("World"));
        this.setBounds(100, 100, 400, 100);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
}
```

Resultat: Wird eine weitere Oberflächendarstellung gewünscht, muss nun nur mehr eine neue Klasse geschrieben werden, welche das Interface `ComponentFactory` implementiert.

2.1.3 Bean-Implementierungen



Gegeben sei folgendes Interface:

```
package appl;

public interface Book {
    public abstract String getIsbn();
    public abstract String getTitle();
    public abstract int getPrice();
}
```

Das Interface kann unterschiedlich implementiert werden. Im Folgenden werden zwei mögliche Implementierungs-Varianten vorgestellt.

Die erste Implementierung ist die "naheliegende":

```
package appl;

public class BookA implements Book {

    private final String isbn;
    private final String title;
    private final int price;

    public BookA(String isbn, String title, int price) {
        this.isbn = isbn;
        this.title = title;
        this.price = price;
    }

    @Override
    public String getIsbn() { return this.isbn; }

    @Override
    public String getTitle() { return this.title; }

    @Override
```

```
public int getPrice()    { return this.price; }

@Override
public String toString() { ... }
}
```

Aber auch folgende, "generische" Implementierung ist möglich:

```
package appl;

import java.util.HashMap;
import java.util.Map;

public class BookB implements Book {

    private final Map<String, Object> entries = new HashMap<>();

    public final String ISBN = "isbn";
    public final String TITLE = "title";
    public final String PRICE = "price";

    public BookB(String isbn, String title, int price) {
        this.entries.put(this.ISBN, isbn);
        this.entries.put(this.TITLE, title);
        this.entries.put(this.PRICE, price);
    }

    @Override
    public String getIsbn() {
        return (String)this.entries.get(this.ISBN);
    }

    @Override
    public String getTitle() {
        return (String)this.entries.get(this.TITLE);
    }

    @Override
    public int getPrice() {
        return (Integer)this.entries.get(this.PRICE);
    }

    @Override
    public String toString() { ... }
}
```

Wir definieren ein "Parallel-Universum" von Factory-Klassen:

```
package appl;

public interface BookFactory {
    public abstract Book createBook(String isbn, String title, int price);
}
```

```
package appl;

public class BookAFactory implements BookFactory {
    @Override
    public Book createBook(String isbn, String title, int price) {
        return new BookA(isbn, title, price);
    }
}
```

```
package appl;

public class BookBFactory implements BookFactory {
    @Override
    public Book createBook(String isbn, String title, int price) {
        return new BookB(isbn, title, price);
    }
}
```

Gegeben sei eine Eingabedatei ("books.txt"):

```
1111 ; Pascal; 10
2222 ; Modula; 20
3333 ; Oberon; 30
```

Aus dieser Eingabe sollen `Book`-kompatible Objekte erzeugt werden – also entweder `BookA`- oder `BookB`-Objekte. Dazu dient ein kleiner `Book`-Parser, dessen `parse`-Methode mit einem `BookFactory`-Parameter parametrisiert ist:

```
package appl;
// ...
public class BookParser {
    public List<Book> parse(InputStream in, BookFactory bookFactory)
        throws Exception {
        final BufferedReader reader =
            new BufferedReader(new InputStreamReader(in));
        final List<Book> books = new ArrayList<Book>();
        String line;
        while ((line = reader.readLine()) != null) {
            final String[] tokens = line.split(";");
            if (tokens.length != 3)
                continue;
            final String isbn = tokens[0].trim();
            final String title = tokens[1].trim();
            final int price = Integer.parseInt(tokens[2].trim());
            final Book book = bookFactory.createBook(isbn, title,
price);
            books.add(book);
        }
        return books;
    }
}
```

Im `BookParser` ist keinerlei Fallunterscheidung erforderlich.

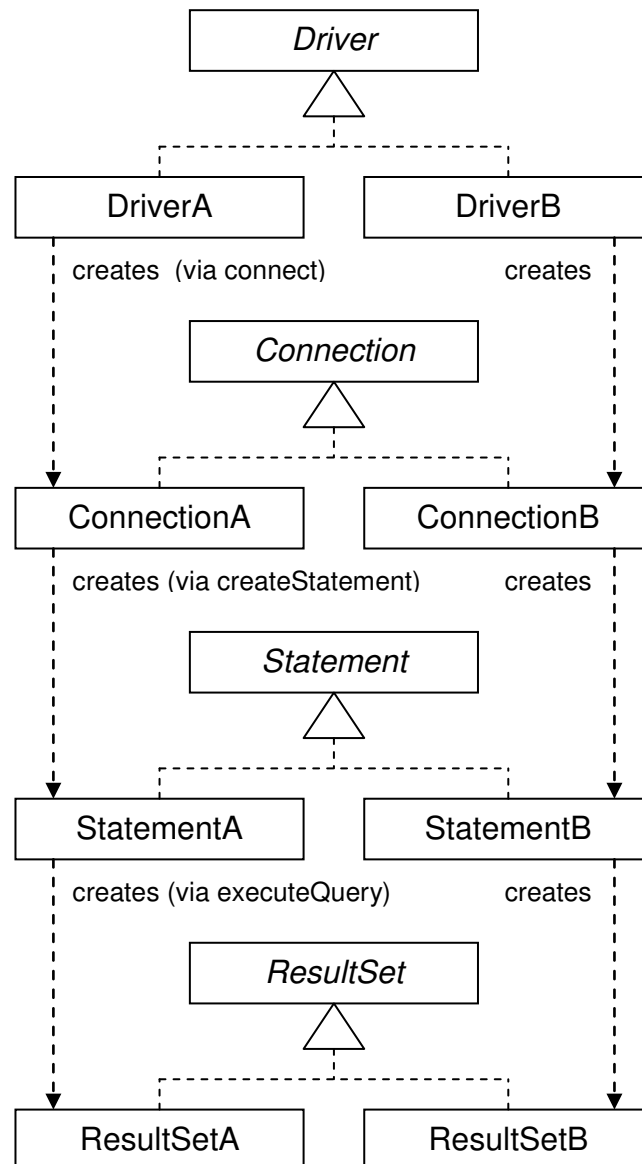
Hier schließlich die Anwendung:

```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        try (InputStream in = new FileInputStream("src/books.txt")) {
            final BookParser parser = new BookParser();
            final List<Book> books = parser.parse(in, new
BookAFactory());
            // final List<Book> books = parser.parse(in, new
BookBFactory());
            books.forEach(System.out::println);
        }
    }
}
```

Aus welchen Gründen könnte man die `BookB`-Implementierung wählen?

2.1.4 JDBC



(A könnte die Derby-Datenbank, B die Oracle-Datenbank sein etc.)

Mittels JDBC können Datenbank-basierte Anwendungen gebaut werden, ohne dabei im Quellcode die konkret genutzte Datenbank als solche anzusprechen. Das JDBC-API enthält eigentlich nur eine einzige Klasse: die Klasse `DriverManager`. Alle anderen Typen, die von JDBC definiert sind, sind Interfaces.

Hier die wichtigsten dieser Interfaces:

```

Driver
Connection
Statement (resp. PreparedStatement und CallableStatement)
ResultSet
  
```

Die Datenbankhersteller implementieren diese Interfaces. Die Derby-Implementierung etwa stellt folgende Klassen bereit:

```
org.apache.derby.impl.jdbc.EmbedDriver  
org.apache.derby.impl.jdbc.EmbedConnection  
org.apache.derby.impl.jdbc.EmbedStatement  
org.apache.derby.impl.jdbc.EmbedResultSet
```

Eine Anwendung, die Derby benutzt, muss aber diese Klassen nicht kennen. Wie funktioniert das?

Zunächst wird die Klasse des verwendeten Drivers geladen. Das kann mit `Class.forName` bewerkstelligt werden (dieser Methode wird nur der Name der Klasse in Form eines Strings übergeben – und dieser String wird typischerweise in einer Properties-Datei hinterlegt (und kann somit jederzeit durch den Namen einer anderen Driver ersetzt werden).

Wenn die Treiberklasse geladen wurde, kann eine Instanz dieser Klasse erzeugt werden (mittels der `newInstance` der Klasse `Class`). Auf diese Treiber-Instanz kann die Methode die im `Driver`-Interface spezifizierte Methode `connect` aufgerufen werden. Ihr wird u.a. die URL der Datenbank übergeben. Diese `connect`-Methode erzeugt ein Objekt einer konkreten Klasse, die das Interface `Connection` implementiert.

Die `Connection` kann dann genutzt werden, um via `getStatement()` eine Instanz einer konkreten Klasse erzeugen zu lassen, die das Interface `Statement` implementiert.

Und mittels dieser `Statement`-Klasse kann schließlich via `executeQuery` eine Instanz einer konkreten Klasse erzeugt werden, die das Interface `ResultSet` implementiert.

Drei Klassen übernehmen hier die Rolle einer Factory:

- Der `Driver` fungiert also als Factory für `Connections`.
- Eine `Connection` fungiert als Factory für `Statements` (resp. `Prepared-` und `CallableStatments`).
- Ein `Statement` fungiert als Factory für `ResultSets`.

Hier eine keine Demo-Anwendung:

```
package appl;

import java.sql.Connection;
import java.sql.Driver;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;

import db.util.appl.Db;

public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        final String driverClassName =
            "org.apache.derby.jdbc.EmbeddedDriver"

        final Driver driver =
            (Driver) Class.forName(driverClassName)
                .newInstance();
        final Properties props = new Properties();
        props.put("user", "user");
        props.put("password", "password");

        final String url =
            "jdbc:derby:../dependencies/derby/data;create=true"

        try (Connection con = driver.connect(url, props)) {
            System.out.println(con.getClass().getName());

            try (final Statement stmt = con.createStatement()) {
                System.out.println(stmt.getClass().getName());

                try (final ResultSet rs =
                    stmt.executeQuery("select title from Book")) {
                    System.out.println(rs.getClass().getName());
                    while (rs.next()) {
                        System.out.println(rs.getString("title"));
                    }
                }
            }
        }
    }
}
```

`Db.aroundAppl()` erzeugt eine Datenbank aufgrund der folgenden `create.sql`:

```
create table book (
  isbn varchar(13),
  title varchar(64),
  price integer,
  primary key (isbn)
);
insert into book values('1111', 'Pascal', 10);
insert into book values('2222', 'Modula', 20);
insert into book values('3333', 'Oberon', 30);
insert into book values('4444', 'Eiffel', 40);
```

Die Ausgaben des obigen Programms:

```
org.apache.derby.impl.jdbc.EmbedConnection40
org.apache.derby.impl.jdbc.EmbedStatement40
org.apache.derby.impl.jdbc.EmbedResultSet40
Pascal
Modula
Oberon
Eiffel
```

By the way: Normalerweise wird die Treiber-Klasse nicht im Anwendungs-Code instanziiert. Und auch die `Connection` wird nicht direkt mittels der `connect`-Methode des Drivers erzeugt. Stattdessen wird die `DriverManager`-Klasse genutzt, um die `Connection` zu erzeugen – so dass normalerweise einiges von dem, was im obigen Programm explizit formuliert wurde, im "Hintergrund" passiert. Siehe hierzu auch den

JDBC-Abschnitt beim Muster "Chain of Responsibility".

Eine `Connection` kann neben einem `Statement` auch ein `PreparedStatement` und ein `CallableStatement` erzeugen. Dies ist im obigen Diagramm aus Platzgründen nicht dargestellt.

Im folgenden wird eine eigene kleine Datenbank gebaut, die statt der Derby-Datenbank verwendet werden kann. Wir implementieren folgende Klassen:

```
simple.SimpleDriver
simple.SimpleConnection
simple.SimpleStatement
simple.SimpleResultSet
```


Das obige Demo-Programm wird auch mit dieser einfachen Datenbank funktionieren. Wir müssen nur den Namen der Treiberklasse und die URL anpassen:

```
final String driverClassName = "simple.SimpleDriver";
```

```
final String url = "jdbc:simple//xyz";
```

Hier die einfache Driver-Implementierung:

```
package simple;
// ...
public class SimpleDriver implements Driver {

    @Override
    public Connection connect(String url, Properties props)
        throws SQLException {
        System.out.println("SimpleDriver.connect(
            " + url + ", " + props + ")");
        if (!url.contains("simple"))
            return null;
        return new SimpleConnection(url);
    }
    // ...
}
```

Tatsächlich enthält das Driver-Interface eine Vielzahl weiterer Methoden, die in der SimpleDriver-Klasse entweder leer implementiert sind resp. einen Default-Wert zurückliefern.

Hier die SimpleConnection:

```
package simple;
// ...
public class SimpleConnection implements Connection {
    private final String url;
    public SimpleConnection (String url) {
        this.url = url;
        // ...
    }

    @Override
    public Statement createStatement() {
        return new SimpleStatement(this);
    }

    // ... (s.o.)
}
```

Die Klasse SimpleStatement:

```
package simple;
// ...
public class SimpleStatement implements Statement {
    private final SimpleConnection con;
    public SimpleStatement(SimpleConnection con) {
        this.con = con;
    }

    @Override
    public ResultSet executeQuery(String sql) {
        return new SimpleResultSet(this);
    }

    // ... (s.o.)
}
```

Und hier schließlich die Klasse SimpleResultSet:

```
package simple;
// ...
public class SimpleResultSet implements ResultSet {

    private final String[] dummyContent = new String[] {
        "Pascal", "Modula", "Oberon", "Eiffel"
    };

    private int index = -1;

    private final SimpleStatement stmt;
    public SimpleResultSet(SimpleStatement stmt) {
        this.stmt = stmt;
    }

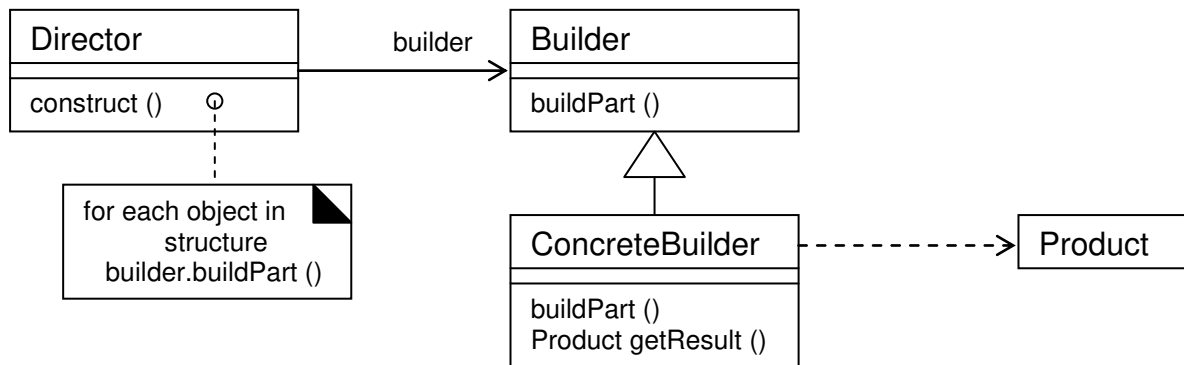
    @Override
    public boolean next() {
        System.out.println("SimpleStatement.next");
        this.index++;
        return this.index < this.dummyContent.length;
    }

    @Override
    public String getString(String columnName) {
        System.out.println("SimpleStatement.getString(" + columnName + ")");
        // ...
        return this.dummyContent[this.index];
    }

    // ... (s.o.)
}
```

2.2 Builder

"Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann." (Gamma, 103)



2.2.1 Problem

Gegeben sei folgende DTD ("topic.dtd"):

```
<!ELEMENT topic (id, (topic*))>
<!ELEMENT id (#PCDATA)>
```

Und folgende XML-Datei ("topic.xml"):

```
<?xml version='1.0'?>
<!DOCTYPE topic SYSTEM "topic.dtd">

<topic>
  <id>Alle Themen</id>
  <topic>
    <id>Programmierung</id>
    <topic>
      <id>Compiler</id>
    </topic>
    <topic>
      <id>Programmiersprachen</id>
      <topic>
        <id>Pascal</id>
      </topic>
      <topic>
        <id>Modula</id>
      </topic>
      <topic>
        <id>Oberon</id>
      </topic>
    </topic>
  </topic>
  <topic>
    <id>Sprachen</id>
    <topic>
      <id>Latein</id>
    </topic>
    <topic>
      <id>Griechisch</id>
    </topic>
  </topic>
</topic>
```

Diese XML-Datei soll eingelesen und zu einem Objekt-Baum transformiert werden.

Der Baum besteht aus Knoten. Diese Knoten können unterschiedlichen Typs sein.

Man könnte z.B. eine (anwendungsspezifische) Klasse `Topic` definieren. Ein `Topic`-Objekt besitzt eine `id` und beliebig viele Kinder (wiederrum vom Typ `Topic`).

Ein `Topic`-Baum, welcher aufgrund des obigen XML-Textes erzeugt werden könnte, würde wie folgt aussehen:

```
id = Alle Themen
  id = Programmierung
    id = Compiler
    id = Programmiersprachen
      id = Pascal
      id = Modula
      id = Oberon
  id = Sprachen
    id = Latein
    id = Griechisch
```

Statt dieser "individuellen", anwendungsspezifischen Implementierung eines Baum-Knotens könnte man aber auch eine "generische" Klasse `Node` definieren (eine Implementierung, die auch für XML-Dokumente anderen Typs (sofern sie keinen "mixed content" benutzen) verwendet werden kann).

Ein `Node`-Baum, der die obige XML-Eingabe repräsentiert, würde ganz anders aussehen:

```
name = topic
  name = id text = Alle Themen
  name = topic
    name = id text = Programmierung
    name = topic
      name = id text = Compiler
    name = topic
      name = id text = Programmiersprachen
        name = topic
          name = id text = Pascal
          name = topic
            name = id text = Modula
            name = topic
              name = id text = Oberon
  name = topic
    name = id text = Sprachen
    name = topic
      name = id text = Latein
    name = topic
      name = id text = Griechisch
```

Hier die Klasse Topic:

```
package appl;
// ...
public class Topic {

    private String id;
    private final List<Topic> children = new ArrayList<>();

    public String getId()          { return this.id; }
    public void setId(String id) { this.id = id; }

    public void add(Topic child) { this.children.add(child); }
    public int size()            { return this.children.size(); }
    public Topic get(int index)  { return this.children.get(index); }

    public void print(String indent) {
        System.out.println(indent + this.id);
        for (final Topic child : this.children)
            child.print(indent + "\t");
    }
}
```

Ein Topic-Objekt besitzt eine `id` und kann beliebig viele Kinder haben, die ihrerseits wiederum Topic-Objekte sind. `size` liefert die Anzahl der Kinder, `get` das `index`-te Kind. `print` gibt den Baum rekursiv aus – wobei die Knoten entsprechend ihrer Tiefe im Baum eingerückt werden (`indent`).

Hier die Klasse Node:

```
package appl;
// ...
public class Node {

    private final String name;
    private String text;
    private final List<Node> children;

    public Node(String name) {
        this.name = name;
        this.children = new ArrayList<>();
    }

    public Node(String name, String text) {
        this.name = name;
        this.text = text;
        this.children = null;
    }

    public String getName()      { return this.name; }
    public String getText()      { return this.text; }

    public void add(Node child) { this.children.add(child); }
    public int size() {
        return this.children == null ? 0 : this.children.size();
    }
    public Node get(int index) {
```

```

        return this.children.get(index);
    }

    public void print(String indent) {
        System.out.print(indent + this.name);
        if (this.text != null)
            System.out.print(" [" + this.text + "]");
        System.out.println();
        if (this.children != null)
            for (final Node child : this.children)
                child.print(indent + "\t");
    }
}

```

Ein `Node` hat einen Namen – den Namen des entsprechenden XML-Elements. Ein Knoten kann einen Text besitzen – sofern es sich um ein inneres XML-Element handelt). Ein Knoten hat *size*-viele Subknoten. Die `get`-Methode liefert den `index`-ten dieser Subknoten zurück. `print` schließlich wird einen `Node`-Baum rekursiv ausgeben.

Man beachte, dass sich die beiden Klassen nicht nur durch ihren Namen unterscheiden: sie unterscheiden sich strukturell (`Node` z.B. hat zwei Konstruktoren; `Topic` hat nur den Default-Konstruktor).

Zum Lesen der XML-Datei benutzen wir den `XMLScanner`, der in der Einleitung bereits beschrieben wurde.

Hier schließlich eine `Application`, welche aufgrund der XML-Datei zwei Bäume erzeugt – einen `Topic`- und einen `Node`-Baum:

```

package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        final String FILENAME = "src/topic.xml";

        final XMLScanner scanner1 = new XMLScanner(
            new FileInputStream(FILENAME));
        final Topic topic = buildTopic(scanner1);
        topic.print("");

        final XMLScanner scanner2 = new XMLScanner(
            new FileInputStream(FILENAME));
        final Node node = buildNode(scanner2);
        node.print("");
    }

    static Topic buildTopic(XMLScanner scanner) throws Exception {
        final Topic topic = new Topic();
        scanner.start("topic");
        final String id = scanner.startTextEnd("id");
        topic.setId(id);
        while (scanner.isStart("topic")) {
            final Topic child = buildTopic(scanner);
            topic.add(child);
        }
    }
}

```

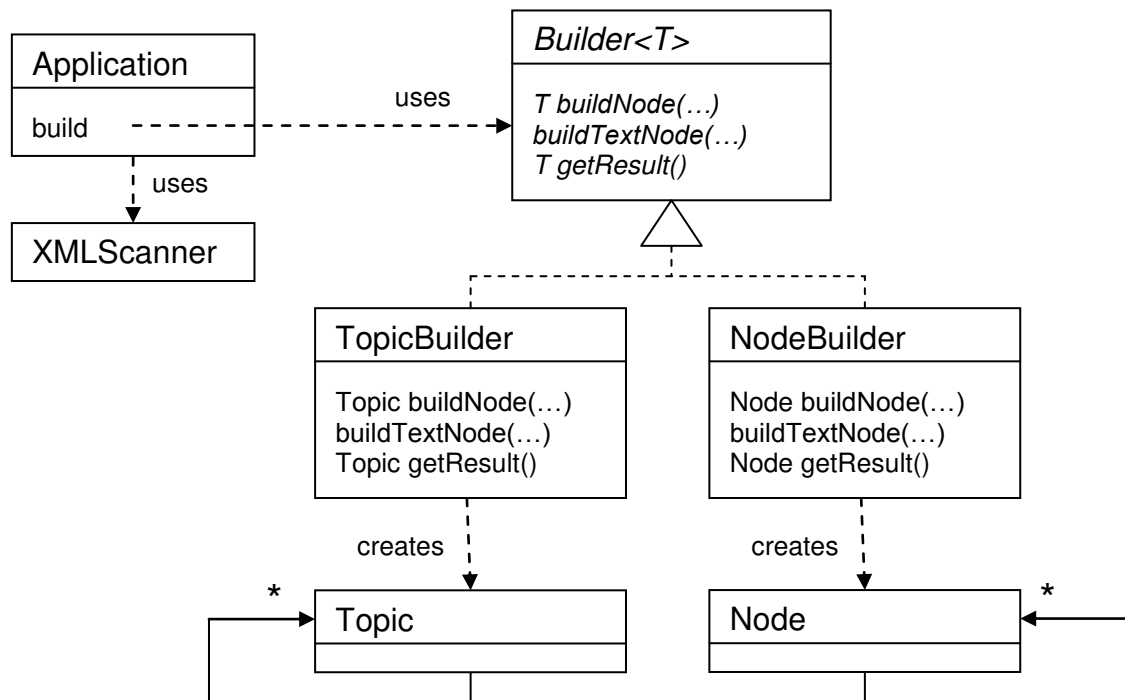
```
        scanner.end("topic");
        return topic;
    }

    static Node buildNode(XMLScanner scanner) throws Exception {
        scanner.start("topic");
        final Node topicNode = new Node("topic");
        final String id = scanner.startTextEnd("id");
        final Node idNode = new Node("id", id);
        topicNode.add(idNode);
        while (scanner.isStart("topic")) {
            final Node child = buildNode(parser);
            topicNode.add(child);
        }
        scanner.end("topic");
        return topicNode;
    }
}
```

Die beiden Methoden `buildTopic` und `buildNode` müssen das XML-Dokument parsen – und beim Parsen dann entsprechende Objekte erzeugen (`Topic`- resp. `Node`-Objekte) und diese zu einem Baum zusammenfügen. Beide Methoden sind rekursiv implementiert. Sie liefern jeweils die Wurzel des erzeugten Baumes zurück.

Besser wäre es, wenn die beiden Aspekte – die Anwendung des Parsers und die Baum-Produktion – voneinander getrennt werden könnten.

2.2.2 Lösung



Die Produktion eines Baumes wird einem Erbauer überlassen, welcher das folgende Interface implementiert:

```
package util;

public interface Builder<T> {
    public abstract T buildNode(String name, T parent);
    public abstract void buildTextNode(String name, String text, T parent);
    public abstract T getResult();
}
```

Ein Erbauer muss übergeordnete Knoten und Text-Knoten (für die inneren XML-Elemente) produzieren können.

Um einen übergeordneten Knoten produzieren zu lassen, wird auf einen Erbauer die Methode `buildNode` aufgerufen werden. Ihr wird neben dem Namen des aktuellen XML-Elements der Vater-Knoten (der bereits zuvor erzeugt wurde!) übergeben. Die Methode muss den produzierten Knoten als Resultat zurückliefern.

Für ein XML-Text-Element wird die Methode `buildTextNode` aufgerufen werden. Dieser Methode wird der Name des XML-Elements und der Text übergeben – und auch hier wieder der bereits zuvor produzierte übergeordnete Knoten. Sie liefert `void` zurück.

Mittels `getResult` muss ein Erbauer nach dem produzierten Baum (dem obersten Knoten dieses Baumes) gefragt werden können.

Hier die Implementierung des Interfaces für Topic-Bäume:

```
package appl;

import util.Builder;

public class TopicBuilder implements Builder<Topic> {
    private Topic result;

    @Override
    public Topic getResult() {
        return this.result;
    }

    @Override
    public Topic buildNode(String name, Topic parent) {
        final Topic topic = new Topic();
        if (parent == null)
            this.result = topic;
        else
            parent.add(topic);
        return topic;
    }

    @Override
    public void buildTextNode(String name, String text, Topic parent) {
        parent.setId(text);
    }
}
```

Und hier die Implementierung für Node-Bäume:

```
package appl;

import util.Builder;

public class NodeBuilder implements Builder<Node> {

    private Node result;

    @Override
    public Node getResult() {
        return this.result;
    }

    @Override
    public Node buildNode(String name, Node parent) {
        final Node node = new Node(name);
        if (parent == null)
            this.result = node;
        else
            parent.add(node);
        return node;
    }

    @Override
    public void buildTextNode(String name, String text, Node parent) {
        parent.add(new Node(name, text));
    }
}
```

Man sieht, dass die Erbauer mit dem Parsen von XML nichts mehr zu tun haben.

Nun kommt es nur noch darauf an, die Methoden eines solchen Erbauers beim Parsen eines XML-Dokuments an den richtigen Stellen aufzurufen:

Die Anwendung erzeugt auch hier wieder einen `Topic`-Baum und einen `Node`-Baum – aber beide mittels des Aufrufs einer einzigen `build`-Methode:

```
package appl;
// ...
import util.Builder;

public class Application {

    public static void main(String[] args) throws Exception {
        final String FILENAME = "src/topic.xml";

        final XMLScanner scanner1 = new XMLScanner(
            new FileInputStream(FILENAME));
        final Topic topic = build(scanner1, new TopicBuilder(), null);
        topic.print("");

        final XMLScanner scanner2 = new XMLScanner(
            new FileInputStream(FILENAME));
        final Node node = build(scanner2, new NodeBuilder(), null);
        node.print("");
    }

    static <T> T build(XMLScanner scanner, Builder<T> builder, T
parent)
        throws Exception {
        final T result = builder.buildNode("topic", parent);
        scanner.start("topic");
        final String id = scanner.startTextEnd("id");
        builder.buildTextNode("id", id, result);
        while (scanner.isStart("topic"))
            build(scanner, builder, result);
        scanner.end("topic");
        return result;
    }
}
```

2.2.3 Ein anderer Builder

Im folgenden wird ein weiteres Konzept vorgestellt, welches allerdings mit dem von Gamma beschriebenen Builder-Pattern nichts zu tun hat – aber eben auch als "Builder" bezeichnet wird. Es geht um die Konstruktion von Objekten, die immutable sind und eine Vielzahl zu initialisierenden Attribute besitzen.

Sei etwa folgende Klasse gegeben:

```
package appl;

public class Foo {
    public final int a;
    public final int b;
    public final int c;
    public final int d;
    public final int e;
    public Foo(int a, int b, int c, int d, int e) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
        this.e = e;
    }
    public Foo(int a, int b, int c, int d) {
        this(a, b, c, d, 0);
    }
    public Foo(int a, int b, int c) {
        this(a, b, c, 0);
    }
    public Foo(int a, int b) {
        this(a, b, 0);
    }
    public Foo(int a) {
        this(a, 0);
    }
    public Foo() {
        this(0);
    }
    @Override
    public String toString() { ... }
}
```

Alle Attribute sind vom Typ int. Dem ersten Konstruktor wird für jedes zu initialisierende Attribut jeweils ein Parameter übergeben. Der Konstruktor ist mehrfach überladen – wird für ein Attribut kein expliziter Parameter angegeben, wird es mit 0 initialisiert. Die mögliche Menge der überladenen Konstruktoren ist natürlich beschränkt.

Hier eine Anwendung dieser Klasse:

```
final Foo foo1 = new Foo(1, 2, 3, 4, 5);
System.out.println(foo1);

final Foo foo2 = new Foo(0, 2, 0, 0, 5);
System.out.println(foo2);
```

Bei der Konstruktion der `Foo`-Objekte müssen wir genau wissen, welche Bedeutung welcher Parameter hat (alle sind dummerweise vom Typ `int` – so dass wir die Bedeutung nicht aufgrund der Typen erraten können...).

Ogleich wir den `Foo`-Konstruktor mehrfach überladen haben, kann die Konstruktion von `Foo`-Objekten zusätzliche Parameter mit dem Wert `0` verlangen (siehe die Konstruktion des zweiten `Foo`-Objekts) – denn wir können natürlich nicht für alle denkbaren Kombinationen von Parametern überladene Konstruktoren bereitstellen.

In solchen Fällen kann ein Muster eingesetzt werden, welches zusätzlich zur Klasse der zu erzeugenden immutable Objects eine Builder-Klasse benutzt, mittels derer die immutable Objects erzeugt werden:

```
package appl;

public class Bar {
    static public class Builder {
        private int a;
        private int b;
        private int c;
        private int d;
        private int e;
        Builder a(int a) {
            this.a = a;
            return this;
        }
        Builder b(int b) {
            this.b = b;
            return this;
        }
        Builder c(int c) {
            this.c = c;
            return this;
        }
        Builder d(int d) {
            this.d = d;
            return this;
        }
        Builder e(int e) {
            this.e = e;
            return this;
        }
        Bar build() {
            return new Bar(this.a, this.b, this.c, this.d, this.e);
        }
    }
    public final int a;
    public final int b;
    public final int c;
    public final int d;
    public final int e;
    private Bar(int a, int b, int c, int d, int e) {
        this.a = a;
        this.b = b;
        this.c = c;
```

```
        this.d = d;
        this.e = e;
    }
    @Override
    public String toString() { ... }
}
```

Die `Bar`-Klasse enthält eine statische innere Klasse namens `Builder`. Nur über deren `build`-Methode kann `Bar`-Objekt erzeugt werden (der Konstruktor von `Bar` ist `private`). Ein `Bar.Builder` wird mittels eines parameterlosen Konstruktors erzeugt. Ein `Bar.Builder` besitzt dieselben Attribute wie die eigentliche `Bar`-Klasse – im Gegensatz zu den Attributen von `Bar` sind sie aber nicht `final`. Mittels der Methoden `a`, `b`, `c` etc. können diese Attribute initialisiert werden. Da all diese Methoden `this` zurückliefern, können sie kaskadierend aufgerufen werden.

Hier eine Anwendung:

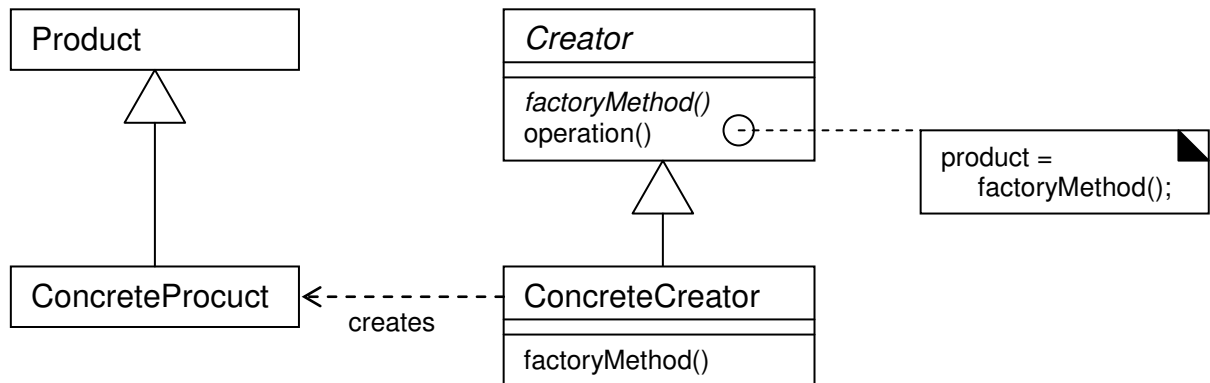
```
final Bar bar1 = new
Bar.Builder().a(1).b(2).c(3).d(4).e(5).build();
System.out.println(bar1);

final Bar bar2 = new Bar.Builder().b(2).c(3).e(5).build();
System.out.println(bar2);
```

Man betrachte hier auch die Klasse `String` (deren Objekte `immutable` sind) und die Klasse `StringBuilder` (resp. `StringBuffer`). Ein `StringBuilder` dient als "Baustelle" für `Strings`.

2.3 Factory Method

"Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren." (Gamma, 115)



2.3.1 Problem

Gegeben sei folgende Datenbank (sieht `create.sql`):

```

create table book (
  isbn varchar(13),
  title varchar(64),
  price integer,
  primary key (isbn)
);

insert into book values('1111', 'Pascal', 10);
insert into book values('2222', 'Modula', 20);
insert into book values('3333', 'Oberon', 30);
insert into book values('4444', 'Eiffel', 40);

```

Die Zeilen der `book`-Tabelle sollen abgebildet werden auf Objekte der Klasse `Book`:

```

package domain;

public class Book {

  private String isbn;
  private String title;
  private int price;

  // setter und getter...

  @Override
  public String toString() { ... }
}

```

Die folgende Anwendung greift zunächst via Primary key auf die Tabelle zu. Dann liest sie schließlich alle Zeilen der Tabelle aus:

```
package appl;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

import db.util.appl.Db;
import domain.Book;

public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
        try (Connection con = DriverManager.getConnection(
            "jdbc:derby:../dependencies/derby/data;create=true",
            "user", "password")) {

            final Book book = findBookByIsbn(con, "2222");
            System.out.println(book);
            System.out.println();

            final List<Book> books = findAllBooks(con);
            books.forEach(System.out::println);
            System.out.println();

        }

        static Book createBook(ResultSet rs) throws Exception {
            final Book book = new Book();
            book.setIsbn(rs.getString("isbn"));
            book.setTitle(rs.getString("title"));
            book.setPrice(rs.getInt("price"));
            return book;
        }

        static Book findBookByIsbn(Connection con, String isbn) {
            final String sql =
                "select isbn, title, price from book where isbn = ?";
            try (PreparedStatement stmt = con.prepareStatement(sql)) {
                stmt.setString(1, isbn);
                try (ResultSet rs = stmt.executeQuery()) {
                    if (!rs.next())
                        return null;
                    final Book book = createBook(rs);
                    return book;
                }
            }
            catch (final Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```



```
static List<Book> findAllBooks(Connection con) {  
    final String sql =  
        "select isbn, title, price from book";  
    try (PreparedStatement stmt = con.prepareStatement(sql)) {  
        try (ResultSet rs = stmt.executeQuery()) {  
            final List<Book> books = new ArrayList<>();  
            while (rs.next()) {  
                final Book book = createBook(rs);  
                books.add(book);  
            }  
            return books;  
        }  
    }  
    catch (final Exception e) {  
        throw new RuntimeException(e);  
    }  
}
```

Die Ausgaben:

Book [isbn=2222, title=Modula, price=20]

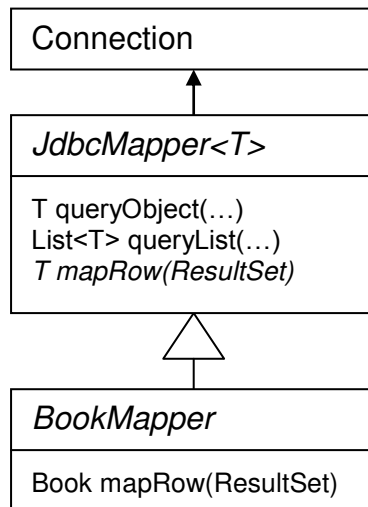
Book [isbn=1111, title=Pascal, price=10]

Book [isbn=2222, title=Modula, price=20]

Book [isbn=3333, title=Oberon, price=30]

Book [isbn=4444, title=Eiffel, price=40]

2.3.2 Lösung



JdbcMapper ist eine abstrakte Basisklasse:

```

package util;
// ...
public abstract class JdbcMapper<T> {

    private final Connection con;

    public JdbcMapper(Connection con) {
        this.con = con;
    }

    public T queryObject(String sql, Object... params) {
        try (PreparedStatement stmt = this.con.prepareStatement(sql)) {
            for (int i = 0; i < params.length; i++)
                stmt.setObject(i + 1, params[i]);
            try (ResultSet rs = stmt.executeQuery()) {
                if (!rs.next())
                    return null;
                final T object = this.mapRow(rs);
                return object;
            }
        } catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }

    public List<T> queryList(String sql, Object... params) {
        try (PreparedStatement stmt = this.con.prepareStatement(sql)) {
            for (int i = 0; i < params.length; i++)
                stmt.setObject(i + 1, params[i]);
            final List<T> list = new ArrayList<T>();
            try (ResultSet rs = stmt.executeQuery()) {
                while (rs.next()) {
                    final T object = this.mapRow(rs);
                    list.add(object);
                }
            }
            return list;
        }
    }
}
  
```

```

        catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }

    protected abstract T mapRow(ResultSet rs) throws Exception;
}

```

Sowohl `queryObject` als auch `queryList` rufen die abstrakte Methode `mapRow` auf – diese muss in abgeleiteten Klassen implementiert werden.

Hier die von `JdbcMapper` abgeleitete Klasse `BookMapper`:

```

package appl;
// ...
public class BookMapper extends JdbcMapper<Book> {

    public BookMapper(Connection con) {
        super(con);
    }

    @Override
    public Book mapRow(ResultSet rs) throws Exception {
        final Book book = new Book();
        book.setIsbn(rs.getString("isbn"));
        book.setTitle(rs.getString("title"));
        book.setPrice(rs.getInt("price"));
        return book;
    }
};

```

Eine Anwendung:

```

package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
        try (Connection con = DriverManager.getConnection(
            "jdbc:derby:../dependencies/derby/data;create=true",
            "user", "password")) {

            final BookMapper bookMapper = new BookMapper(con);

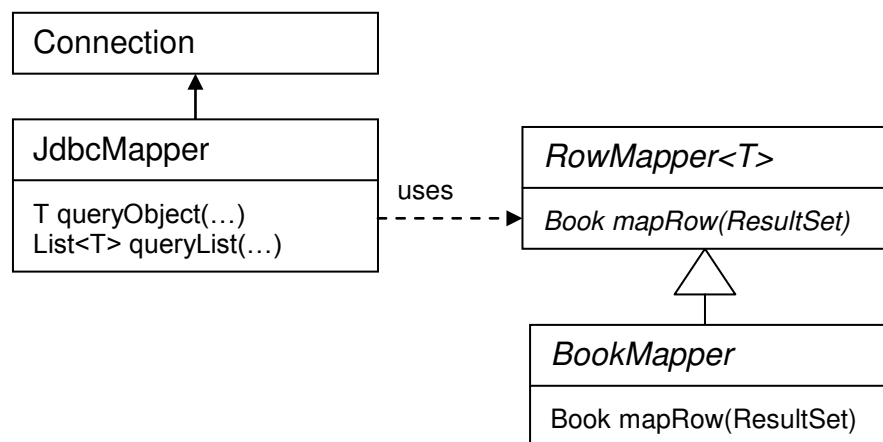
            final Book book = bookMapper.queryObject(
                "select isbn, title, price from book where isbn = ?",
                "2222");
            System.out.println(book);
            System.out.println();

            final List<Book> books = bookMapper.queryList(
                "select isbn, title, price from book");
            books.forEach(System.out::println);
            System.out.println();

        }
    }
}

```

2.3.3 Delegation via Interface



Wie beginnen mit einem Interface:

```

package util;

import java.sql.ResultSet;

public interface RowMapper<T> {
    public abstract T mapRow(ResultSet rs) throws Exception;
}
  
```

Die Klasse JdbcMapper benutzt einen RowMapper – und ist nun instantiierbar:

```

package util;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

public class JdbcMapper {

    private final Connection con;

    public JdbcMapper (Connection con) {
        this.con = con;
    }

    public <T> T queryObject(String sql, RowMapper<T> rowMapper,
        Object... params) {
        try (PreparedStatement stmt = this.con.prepareStatement(sql)) {
            for (int i = 0; i < params.length; i++)
                stmt.setObject(i + 1, params[i]);
            try (ResultSet rs = stmt.executeQuery()) {
                if (!rs.next())
                    return null;
                final T object = rowMapper.mapRow(rs);
                return object;
            }
        }
    }
}
  
```

```

        catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }

    public <T> List<T> queryList(String sql, RowMapper<T> rowMapper,
        Object... params) {
        try (PreparedStatement stmt = this.con.prepareStatement(sql)) {
            for (int i = 0; i < params.length; i++)
                stmt.setObject(i + 1, params[i]);
            final List<T> list = new ArrayList<T>();
            try (ResultSet rs = stmt.executeQuery()) {
                while (rs.next()) {
                    final T object = rowMapper.mapRow(rs);
                    list.add(object);
                }
            }
            return list;
        }
    }
    catch (final Exception e) {
        throw new RuntimeException(e);
    }
}

```

Eine Anwendung:

```

package appl;
// ...
public class Application {

    static RowMapper<Book> bookRowMapper = new RowMapper<Book>() {
        @Override
        public Book mapRow(ResultSet rs) throws Exception {
            final Book book = new Book();
            book.setIsbn(rs.getString("isbn"));
            book.setTitle(rs.getString("title"));
            book.setPrice(rs.getInt("price"));
            return book;
        }
    };

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
        try (Connection con = DriverManager.getConnection(
            "jdbc:derby:../dependencies/derby/data;create=true",
            "user", "password")) {

            final JdbcMapper mapper = new JdbcTemplate(con);
            final Book book = mapper.queryObject(
                "select isbn, title, price from book where isbn = ?",
                bookRowMapper,
                "2222");
            System.out.println(book);
            System.out.println();

            final List<Book> books = mapper.queryList(
                "select isbn, title, price from book",

```

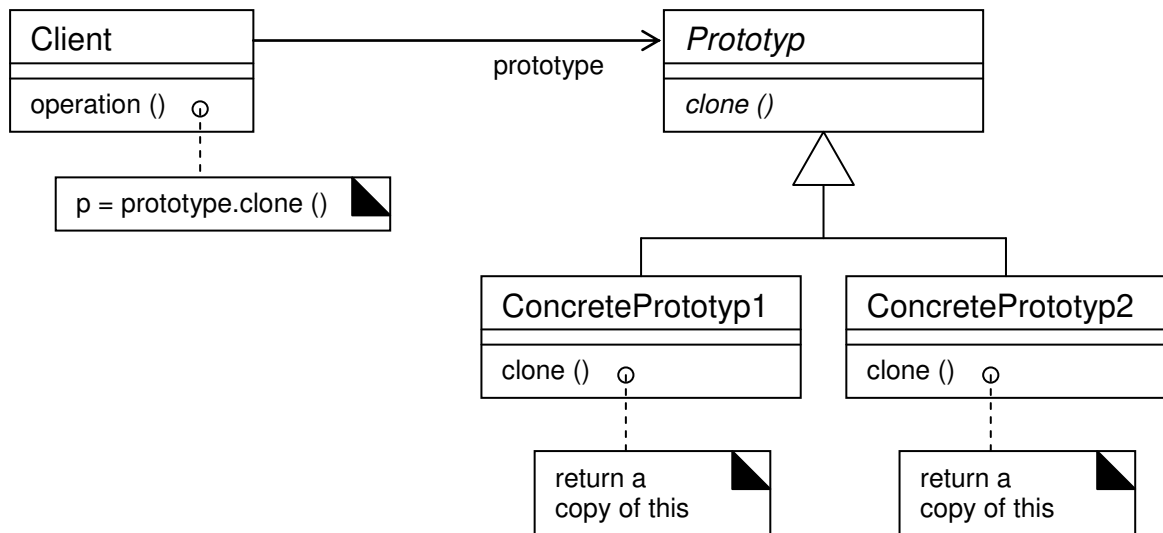
```
        bookRowMapper);  
        books.forEach(System.out::println);  
        System.out.println();  
    }  
}  
}
```

Der in der Application verwendete RowMapper kann auch etwas einfacher formuliert werden – als Lambda-Ausdruck:

```
static RowMapper<Book> bookRowMapper2 = rs -> {  
    final Book book = new Book();  
    book.setIsbn(rs.getString("isbn"));  
    book.setTitle(rs.getString("title"));  
    book.setPrice(rs.getInt("price"));  
    return book;  
};
```

2.4 Prototype

"Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototypen." (Gamma, 127)



2.4.1 Problem

Die Präferenzen eines Benutzers seien in einer Properties-Datei beschrieben (`preferences.properties`):

```
foreground = black
background = white
fontName = arial
fontSize = 16
```

Präferenzen werden von Objekten der Klasse `Preferences` repräsentiert:

```
package util;

public class Preferences {

    private String background;
    private String foreground;
    private String fontName;
    private int fontSize;

    // setter, getter...

    @Override
    public String toString() { ... }
}
```

Eine Anwendung könnte eine gegebene Property-Datei einlesen und die Standard-Werte für die Präferenzen aus eben dieser Datei entnehmen.

```
package appl;

import java.io.FileInputStream;
import java.util.Properties;
import util.Preferences;

public class Application {

    public static void main(String[] args) throws Exception {
        final Properties props = new Properties();
        props.load(new FileInputStream("src/preferences.properties"));

        final Preferences p1 = new Preferences();
        p1.setBackground(props.getProperty("background"));
        p1.setForeground(props.getProperty("foreground"));
        p1.setFontName(props.getProperty("fontName"));
        p1.setFontSize(20);
        System.out.println(p1);

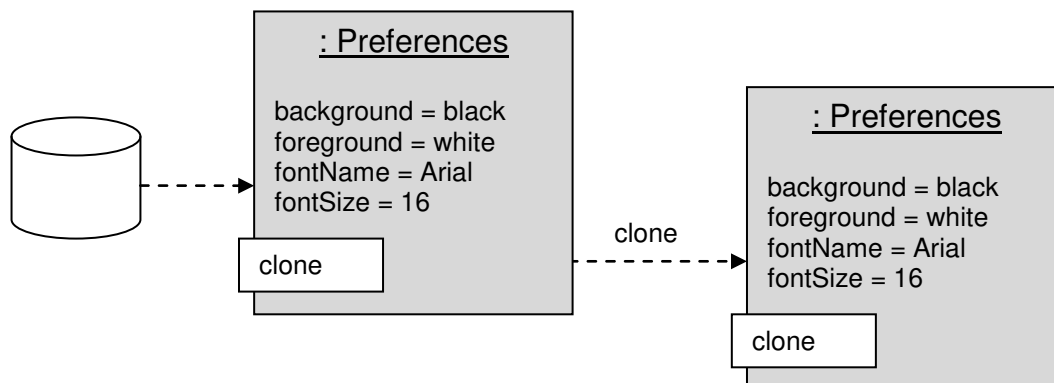
        final Preferences p2 = new Preferences();
        p2.setBackground("grey");
        p2.setForeground("blue");
        p2.setFontName(props.getProperty("fontName"));
        p2.setFontSize(Integer.parseInt(
            props.getProperty("fontSize")));
        System.out.println(p2);
    }
}
```


Hier die Ausgaben des Programms:

```
Preferences [  
  background=white, foreground=black, fontName=arial, fontSize=20]  
Preferences [  
  background=grey, foreground=blue, fontName=arial, fontSize=16]
```

Es wäre natürlich schön, wenn die von der Property-Datei zu übernehmenden Präferenzen nicht jeweils einzeln in das neue `Preferences`-Objekt kopiert werden müssten – wenn ein `Preferences`-Objekt explizit als "Vorlage" für neue `Preferences`-Objekte fungieren könnte.

2.4.2 Lösung



Man kann die `Preferences`-Klasse mit einer `clone`-Methode ausstatten (und dabei das Interface `Cloneable` implementieren):

```
package util;
// ...
public class Preferences implements Cloneable {

    private String background;
    private String foreground;
    private String fontName;
    private int fontSize;

    public Preferences(String filename) throws IOException {
        final Properties props = new Properties();
        props.load(new FileInputStream(filename));
        this.background = props.getProperty("background");
        this.foreground = props.getProperty("foreground");
        this.fontName = props.getProperty("fontName");
        this.fontSize =
        Integer.parseInt(props.getProperty("fontSize"));
    }

    @Override
    public Preferences clone() {
        try {
            return (Preferences) super.clone();
        }
        catch (final CloneNotSupportedException e) {
            throw new RuntimeException(e);
        }
    }

    // setter, getter, toString...
}
```

Die `clone`-Methode erstellt eine Kopie des Originals und liefert diese Kopie als `Object` zurück.

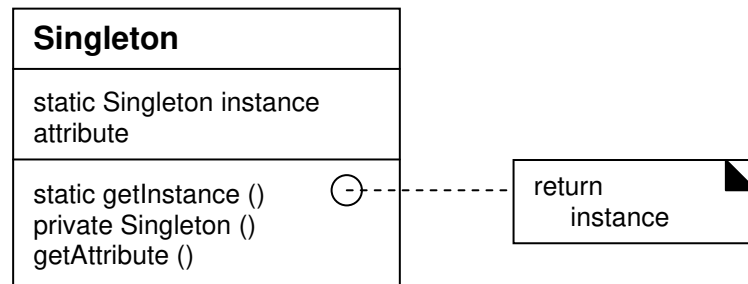
Die Anwendung sieht dann wesentlich "aufgeräumter" aus:

```
package appl;  
  
import util.Preferences;  
  
public class Application {  
  
    public static void main(String[] args) throws Exception {  
  
        final Preferences p0 = new  
Preferences("src/preferences.properties");  
        System.out.println(p0);  
  
        final Preferences p1 = p0.clone();  
        p1.setFontSize(20);  
        System.out.println(p1);  
  
        final Preferences p2 = p0.clone();  
        p2.setBackground("grey");  
        p2.setForeground("blue");  
        System.out.println(p2);  
    }  
}
```

Das erste (via `p0` referenzierte) `Preferences`-Objekt dient nun als Prototyp für die Erzeugung neuer `Preferences`-Objekte. Nach dieser Erzeugung müssen nur diejenigen Properties neu gesetzt werden, die vom "Standard" abweichen.

2.5 Singleton

"Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit." (Gamma, 139)



2.5.1 Problem

Sterne können durch folgende Klasse beschrieben werden:

```
package appl;

public class Star {
    public final String name;
    public Star(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Star [name=" + this.name + "]";
    }
}
```

Und Universum-Objekte durch folgende Klasse:

```
package appl;
// ...
public class Universe {
    private final List<Star> stars = new ArrayList<>();
    public void add(Star star) {
        this.stars.add(star);
    }
    public int size() {
        return this.stars.size();
    }
    public Star get(int index) {
        return this.stars.get(index);
    }
}
```

Eine Anwendung könnte diese beiden Klassen wie folgt nutzen:

```
package appl;

public class Application {

    public static void main(String[] args) throws Exception {
        final Universe u = new Universe();
        u.add(new Star("Morgenstern"));
        u.add(new Star("Abendstern"));
        for (int i = 0; i < u.size(); i++)
            System.out.println(u.get(i));

        final Universe u2 = new Universe();
        u2.add(new Star("Castor"));
        for (int i = 0; i < u2.size(); i++)
            System.out.println(u2.get(i));
    }
}
```

Es sollte zwar viele Sterne, aber eigentlich nur ein einziges Universum geben können...

2.5.2 Lösung

Die Klasse `Universe` wird mit einem privaten Konstruktor ausgestattet (somit ist eine Erzeugung von `Universe`-Objekten außerhalb der `Universe`-Klasse nicht möglich). Sie wird weiterhin ausgestattet mit einer privaten statischen (also "globalen") Referenz, welche mit einem `Universe`-Objekt initialisiert wird (beim Laden der Klasse). Eine öffentliche statische (also ebenfalls "globale") Property erlaubt den lesenden Zugriff auf diese Referenz. Somit ist sichergestellt, dass es nur ein einziges Universum gibt – und es ist gleichzeitig sichergestellt, dass überall auf dieses einzige Exemplar zugegriffen werden kann (über die statische Property).

```
package appl;
// ...
public class Universe {

    private static Universe instance = new Universe();
    public static Universe getInstance() {
        return instance;
    }
    private Universe() {
    }

    private final List<Star> stars = new ArrayList<>();
    public void add(Star star) {
        this.stars.add(star);
    }
    public int size() {
        return this.stars.size();
    }
    public Star get(int index) {
        return this.stars.get(index);
    }
}
```

Hier die neue Anwendung:

```
package appl;

public class Application {

    public static void main(String[] args) throws Exception {
        final Universe u = Universe.getInstance();
        u.add(new Star("Morgenstern"));
        u.add(new Star("Abendstern"));
        for (int i = 0; i < u.size(); i++)
            System.out.println(u.get(i));

        final Universe u2 = Universe.getInstance();
        System.out.println(u == u2);
        u2.add(new Star("Castor"));
        for (int i = 0; i < u2.size(); i++)
            System.out.println(u2.get(i));
    }
}
```

Die beiden `getInstance`-Aufrufe liefern dieselbe Referenz zurück. Die Ausgaben der obigen Anwendung:

```
Star [name=Morgenstern]
Star [name=Abendstern]
true
Star [name=Morgenstern]
Star [name=Abendstern]
Star [name=Castor]
```

2.5.3 Lazy Creation

Angenommen, die Erzeugung des Universums ist kostspielig. Dann sollte es erst dann (und nur dann!) erzeugt werden, wenn es das erste Mal benötigt wird (und eben noch nicht beim Laden der Klasse):

```
package appl;
// ...
public class Universe {

    private static Universe instance;
    synchronized public static Universe getInstance() {
        if (instance == null)
            instance = new Universe();
        return instance;
    }
    private Universe() {
    }

    private final List<Star> stars = new ArrayList<>();
    public void add(Star star) {
        this.stars.add(star);
    }
    public int size() {
        return this.stars.size();
    }
    public Star get(int index) {
        return this.stars.get(index);
    }
}
```

Man beachte das `synchronized`-Schlüsselwort!

2.5.4 Serialisierungs-Problem

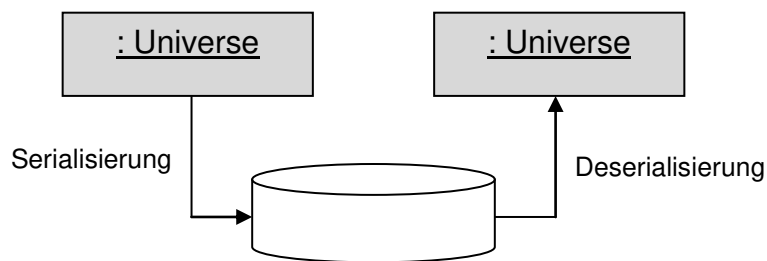
Eigentlich ist es natürlich "sinnlos", ein Universum serialisieren zu wollen. Wird probieren es trotzdem einmal aus.

Wir erweitern zunächst die Klassen `Universe` und `Star` um die Implementierung des `Serializable`-Interfaces (welches bekanntlich ein reines Marker-Interface ist):

```
public class Universe implements Serializable { ... }
```

```
public class Star implements Serializable { ... }
```

Wir serialisieren in eine Datei – um dann sofort wieder zu deserialisieren:



```
package appl;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Application {

    public static void main(String[] args) throws Exception {
        final Universe u = Universe.getInstance();
        u.add(new Star("Morgenstern"));
        u.add(new Star("Abendstern"));
        for (int i = 0; i < u.size(); i++)
            System.out.println(u.get(i));

        try(ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("zzz.ser"))) {
            out.writeObject(u);
        }

        try(ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("zzz.ser"))) {
            final Universe u2 = (Universe)in.readObject();
            System.out.println(u == u2);
        }
    }
}
```

Die Ausgaben zeigen, dass es nun zwei(!!!) `Universe`-Objekte gibt. Das ist nicht im Sinne eines Singletons...

Wie kann man diese "Verdopplung" vermeiden?

2.5.5 ReadResolve

Wir erweitern die Klasse `Universe` um eine Methode `readResolve` – eine Methode, welche, sofern sie existiert, bei der Deserialisierung aufgerufen wird:

```
package appl;

import java.io.ObjectStreamException;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class Universe implements Serializable {

    private static Universe instance;
    synchronized public static Universe getInstance() {
        if (instance == null)
            instance = new Universe();
        return instance;
    }
    private Universe() {
    }

    // ...

    private Object readResolve() throws ObjectStreamException {
        System.out.println("in readResolve: " + (this == instance));
        return instance;
    }
}
```

Wir benutzen dieselbe Anwendung wie im letzten Beispiel:

```
public static void main(String[] args) throws Exception {
    final Universe u = Universe.getInstance();
    u.add(new Star("Morgenstern"));
    u.add(new Star("Abendstern"));
    for (int i = 0; i < u.size(); i++)
        System.out.println(u.get(i));

    try(ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("zzz.ser"))) {
        out.writeObject(u);
    }

    try(ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("zzz.ser"))) {
        final Universe u2 = (Universe)in.readObject();
        System.out.println(u == u2);
    }
}
```

Die Ausgaben:

```
Star [name=Morgenstern]
Star [name=Abendstern]
in readResolve: false
true
```

Das serialisierte `Universe` hat sich bei seiner Deserialisierung durch dasjenige `Universe` "ersetzt", welches ja ohnehin bereits existiert...

2.5.6 Singletons als einfache enums

Vielleicht sollte man Singletons einfach mittels `enums` implementieren, welche jeweils genau eine einzige Ausprägung haben (dann hat man auch keinerlei Probleme mit der Serialisierung / Deserialisierung):

```
package appl;
// ...
public enum Universe {
    INSTANCE;

    private final List<Star> stars = new ArrayList<>();

    public void add(Star star) { ... }
    public int size() { ... }
    public Star get(int index) { ... }
}
```

Eine Anwendung:

```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        final Universe u = Universe.INSTANCE;
        u.add(new Star("Morgenstern"));
        u.add(new Star("Abendstern"));
        for (int i = 0; i < u.size(); i++)
            System.out.println(u.get(i));

        try(ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("zzz.ser"))) {
            out.writeObject(u);
        }

        try(ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("zzz.ser"))) {
            final Universe u2 = (Universe)in.readObject();
            System.out.println(u == u2);
        }
    }
}
```

Die Ausgaben:

```
Star [name=Morgenstern]
Star [name=Abendstern]
true
```

2.5.7 Tripleton

Die Grundfarben rot, grün und blau sollen als Objekte einer Color-Klasse definiert werden. Wir wollen zusichern, dass die Color-Klasse genau drei Instanzen hat.

Hier die erste Version:

```
package appl;

public class Color1 implements Comparable<Color1> {

    public static final Color1 RED = new Color1(0, "RED");
    public static final Color1 GREEN = new Color1(1, "GREEN");
    public static final Color1 BLUE = new Color1(2, "BLUE");

    private final int ordinal;
    private final String name;

    private Color1(int ordinal, String name) {
        this.ordinal = ordinal;
        this.name = name;
    }

    public final int ordinal() { return this.ordinal; }
    public final String name() { return this.name; }

    @Override
    public String toString() { return this.name; }

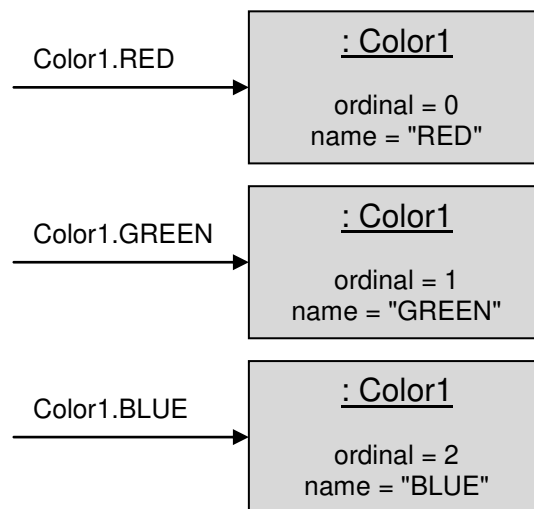
    public static Color1[] values() {
        return new Color1[] { RED, GREEN, BLUE };
    }

    public static Color1 valueOf(String name) {
        for (final Color1 c : values())
            if (c.name().equals(name))
                return c;
        return null;
    }

    @Override
    public final int compareTo(Color1 other) {
        return this.ordinal - other.ordinal;
    }
}
```

Der Konstruktor ist private.

Es gibt genau drei statische konstante Referenzen, welche auf `Color1`-Objekte zeigen.



Jedes `Color1`-Objekt hat einen Ordinalwert und einen Namen. Diese können mittels der finalen Methoden `ordinal()` und `name()` ausgelesen werden. Zudem existiert eine statische Methode, welche einen Array aller `Color1`-Objekt liefert und eine weitere statische Methode, welche aufgrund eines Namens das dazugehörige `Color1`-Objekt liefert.

Schließlich ist die Klasse `Comparable` und überschreibt `toString`.

Hier eine Benutzung der Klasse:

```
static void demo1() {  
    final Color1 blue = Color1.BLUE;  
    System.out.println(blue);  
    System.out.println(blue.ordinal());  
    System.out.println(blue.name());  
    System.out.println(Arrays.toString(Color1.values()));  
    System.out.println(Color1.valueOf("GREEN"));  
    System.out.println(Color1.BLUE.compareTo(Color1.RED));  
}
```

Die Ausgaben:

```
BLUE  
2  
BLUE  
[RED, GREEN, BLUE]  
GREEN  
2
```

Hier eine zweite Version dieser Klasse, welche komplett äquivalent zu `Color1` ist – aber viel rascher notiert:

```
package appl;

public enum Color2 {
    RED, GREEN, BLUE
}
```

Der Compiler erzeugt aufgrund des hier definierten `enum`-Typs eine Klasse, die exakt dieselben Elemente besitzt wie `Color1`.

Das zeigt auch die Verwendung dieses Typs:

```
static void demo2() {
    final Color2 blue = Color2.BLUE;
    System.out.println(blue);
    System.out.println(blue.ordinal());
    System.out.println(blue.name());
    System.out.println(Arrays.toString(Color2.values()));
    System.out.println(Color2.valueOf("GREEN"));
    System.out.println(Color1.BLUE.compareTo(Color1.RED));
}
```

Eine weitere Version, welche nun zusätzlich weitere Instanzvariablen und Instanz-Methoden definiert – und das Interface `Displayable` implementiert (dieses Interface hat eine einzige Methode: `display`):

```
package appl;

public class Color3 implements Comparable<Color3>, Displayable {
    public static final Color3 RED = new Color3(0, "RED", 255, 0, 0);
    public static final Color3 GREEN = new Color3(1, "GREEN", 0, 255, 0);
    public static final Color3 BLUE = new Color3(2, "BLUE", 0, 0, 255);

    private final int ordinal;
    private final String name;
    private final int r;
    private final int g;
    private final int b;

    private Color3(int ordinal, String name, int r, int g, int b) {
        this.ordinal = ordinal;
        this.name = name;
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public final int ordinal() { return this.ordinal; }
    public final String name() { return this.name; }
    public int r() { return this.r; }
    public int g() { return this.g; }
    public int b() { return this.b; }

    @Override
```

```
public String toString() {  
    return this.name +  
        " (" + this.r + ", " + this.g + ", " + this.b + ")";  
}  
  
public static Color3[] values() { ... }  
public static Color3 valueOf(String name) { ... }  
  
@Override  
public int compareTo(Color3 other) { ... }  
  
@Override  
public void display() {  
    System.out.println(this.r + " " + this.g + " " + this.b);  
}  
}
```

Die Klasse definiert drei weitere Instanzvariablen (r, g, b), erweitert den Konstruktor um die Initialisierung dieser drei Variablen und bietet entsprechende Methoden zum Auslesen dieser Variablen an. Diese Werte werden nun auch von `toString` zurückgegeben.

Hier eine Verwendung dieser Klasse:

```
static void demo3() {  
    System.out.println(Color3.RED);  
    System.out.println(Color3.GREEN);  
    System.out.println(Color3.BLUE);  
}
```

Die Ausgaben:

```
RED (255, 0, 0)  
GREEN (0, 255, 0)  
BLUE (0, 0, 255)
```


Das Ganze hätte man auch einfacher haben können (auch `enum`-Typen können erweitert werden):

```
package appl;

public enum Color4 implements Displayable{
    RED(255, 0, 0),
    GREEN(0, 255, 0),
    BLUE(0, 0, 255);

    private final int r;
    private final int g;
    private final int b;

    private Color4(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public int r() { return this.r; }
    public int g() { return this.g; }
    public int b() { return this.b; }

    @Override
    public String toString() {
        return this.name() +
            " (" + this.r + ", " + this.g + ", " + this.b + ")";
    }

    @Override
    public void display() {
        System.out.println(this.r + " " + this.g + " " + this.b);
    }
}
```

Ein `enum`-Typ kann also zusätzliche Attribute und Methoden definieren. Die `toString`-Methode ist überschreibbar. Und ein `enum`-Typ kann auch weitere Interfaces implementieren (er kann allerdings nicht von einer Klasse abgeleitet werden).

2.5.8 Chess

Ein Schachbrett hat 64 Positionen. Positionen soll mittels `Position`-Objekten beschrieben werden:

```
package appl;

public class Position {
    public final int x;
    public final int y;
    private Position(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public static final int N = 8;
    private static Position[][] positions = new Position[N][N];
    static {
        for (int x = 0; x < N; x++)
            for (int y = 0; y < N; y++)
                positions[x][y] = new Position(x, y);
    }
    public static Position get(int x, int y) {
        return Position.positions[x][y];
    }
    @Override
    public String toString() {
        return this.getClass().getSimpleName() +
            " [x=" + this.x + ", y=" + this.y + "]";
    }
}
```

Die Klasse sichert zu, dass es niemals die 65-te Position geben wird. Und `Position`-Objekten kann via Referenz-Vergleich verglichen werden. Die `equals`- und `hashCode`-Methoden müssen also nicht überschrieben werden.

Eine Anwendung:

```
package appl;

public class Application {

    public static void main(String[] args) {
        final Position p1 = Position.get(3, 4);
        final Position p2 = Position.get(3, 4);
        System.out.println(p1);
        System.out.println(p2.x + " " + p2.y);
        System.out.println(p1 == p2);
    }
}
```

Die Ausgaben:

```
Position [x=3, y=4]
3 4
true
```

2.5.9 Runtime und Toolkit

Abschließend ein kleines Anwendungs-Beispiel für zwei Singleton-Klassen der Standardbibliothek:

```
package appl;

import java.awt.Toolkit;

public class Application {

    public static void main(String[] args) {

        final Runtime runtime = Runtime.getRuntime();
        System.out.println(runtime.totalMemory());
        System.out.println(runtime.freeMemory());
        runtime.addShutdownHook(new Thread(
            () -> System.out.println("after main")));

        final Toolkit toolkit = Toolkit.getDefaultToolkit();
        System.out.println(toolkit.getScreenSize());
        toolkit.beep();
        System.out.println("end of main");
    }
}
```

Die Ausgaben:

```
64487424
63144976
java.awt.Dimension[width=1600,height=900]
end of main
after main
```

3

Strukturmuster

3.1	Adapter	3-4
3.1.1	Problem	3-5
3.1.2	Lösung	3-7
3.1.3	Klassen-Adapter	3-9
3.1.4	Listener-Registratur	3-11
3.1.5	Runnable	3-14
3.1.6	Lambdas	3-17
3.1.7	Ad-hoc-Lambdas	3-18
3.1.8	Swing: ActionListener	3-19
3.1.9	Swing: TreeModel	3-23
3.2	Bridge	3-27
3.2.1	Problem	3-28
3.2.2	Lösung	3-33
3.2.3	Figures und Drawers	3-36
3.3	Decorator	3-39
3.3.1	Problem	3-40
3.3.2	Lösung	3-44
3.3.3	Reader	3-46
3.4	Facade	3-50
3.4.1	Problem	3-51
3.4.2	Lösung	3-53

3.5	Flyweight	3-56
3.5.1	Problem	3-57
3.5.2	Lösung	3-60
3.5.3	Ein String-Cache.....	3-63
3.6	Composite	3-65
3.6.1	Problem	3-66
3.6.2	Lösung	3-70
3.6.3	Expressions	3-72
3.7	Proxy	3-77
3.7.1	Problem	3-78
3.7.2	Lösung	3-79
3.7.3	InvocationHandler	3-82
3.7.4	DynamicProxy.....	3-86
3.7.5	SimpleInvocationHandler	3-88

3 Strukturmuster

Strukturmuster befassen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden.

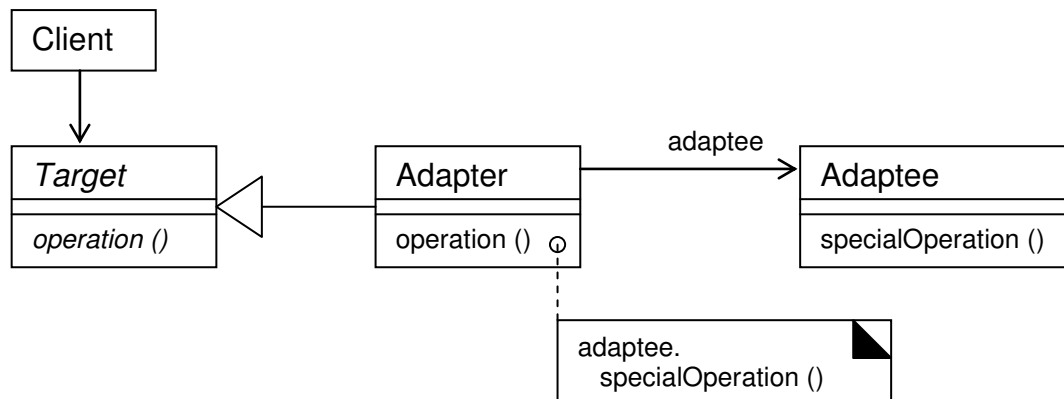
Klassenbasierte Strukturmuster benutzen Vererbung, um Schnittstellen oder Implementierungen zusammenzufügen....

Objektbasierte Strukturmuster hingegen beschreiben [...] Mittel und Wege, Objekte zusammenzufügen, um neue Funktionalität zu gewinnen. Die zusätzliche Flexibilität der Objektkomposition ergibt sich aus der Möglichkeit, das Kompositionsgefüge zur Laufzeit zu ändern, was mit statischer Klassenkomposition nicht möglich ist."

(Gamma, 149)

3.1 Adapter

"Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster lässt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten dazu nicht in der Lage wären." (Gamma, 151)



3.1.1 Problem

Eine Heizung könnte von einem Thermostat gesteuert werden (Brenner einschalten, Brenner ausschalten). Natürlich sollte auch ein Kühlschrank von einem Thermostat gesteuert werden können. Das Thermostat sollte also den genauen Typ des von ihr zu steuernden Objekts nicht kennen. Deshalb definieren die Thermostatbauer zunächst folgendes Interface:

```
package theo;

public interface ThermostatListener {
    public abstract void minAlarm();
    public abstract void maxAlarm();
}
```

Einem Thermostat muss dann (per Konstruktor) ein Objekt übergeben werden, dessen Klasse dieses Interface implementiert. In seiner run-Methode misst ein Thermostat dann permanent die aktuelle Temperatur und informiert den ThermostatListener immer dann, wenn es zu kalt bzw. zu warm geworden ist. Diese run-Methode kann hier natürlich nur simuliert werden:

```
package theo;

public class Thermostat {

    private final ThermostatListener listener;

    public Thermostat(ThermostatListener listener) {
        this.listener = listener;
    }

    public void run() {
        // es ist zu kalt geworden
        this.listener.minAlarm();

        // es ist zu warm geworden
        this.listener.maxAlarm();
    }
}
```

Die Klasse Heizung könnte dann das Interface ThermostatListener wie folgt implementieren:

```
package heinz;

import theo.ThermostatListener;

public class Heizung implements ThermostatListener {

    @Override
    public void minAlarm() {
        System.out.println("Brenner ein");
    }
}
```



```

@Override
public void maxAlarm() {
    System.out.println("Brenner aus");
}
}

```

Auch eine Klasse `Kühlschrank` könnte dieses Interface implementieren (natürlich anders!).

Eine `Heizung` könnte dann auf folgende Weise mit einem `Thermostat` verbunden werden:

```

package appl;

import heinz.Heizung;
import theo.Thermostat;

public class Application {
    public static void main(String[] args) {
        final Heizung h = new Heizung();
        final Thermostat t = new Thermostat(h);
        t.run();
    }
}

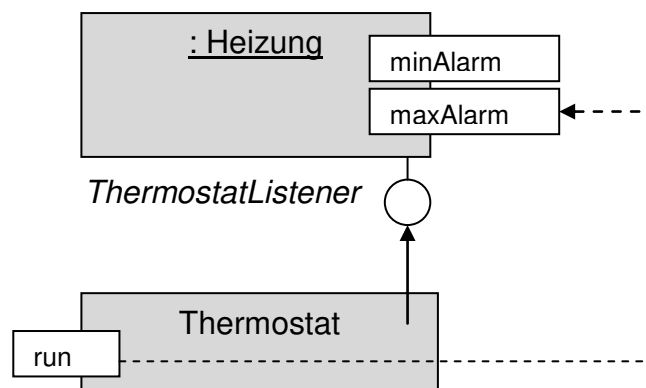
```

Die Ausgaben:

Brenner aus

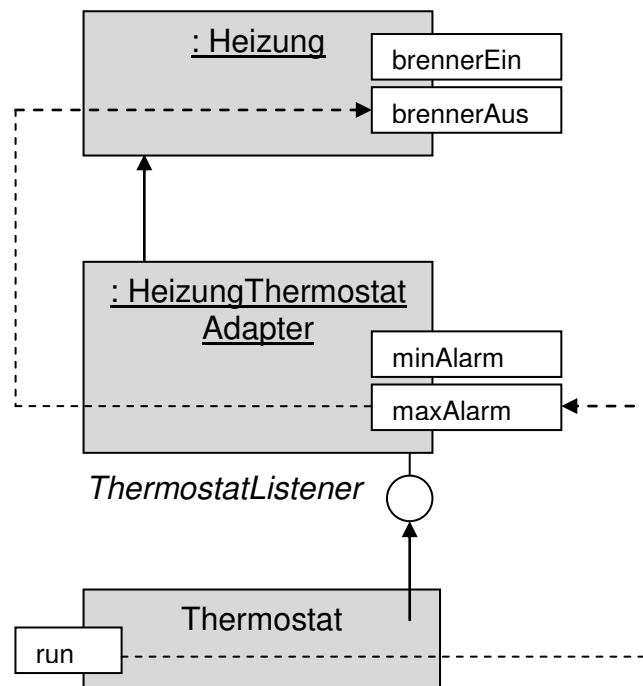
Brenner ein

Ein Objektdiagramm:



Angenommen aber, die Heizungsbauer wissen nichts von den Thermostatbauern – sie kennen also das Interface `ThermostatListener` nicht und können es daher auch in ihrer `Heizung`-Klasse nicht implementieren. Dann kann eine `Heizung` natürlich nicht an den `Thermostat`-Konstruktor übergeben werden.

3.1.2 Lösung



Hier die neue Klasse `Heizung` (die vom `ThermostatListener` nicht weiß):

```
package heinz;

public class Heizung {

    public void brennerEin() {
        System.out.println("Brenner ein");
    }

    public void brennerAus() {
        System.out.println("Brenner aus");
    }

}
```

Zwischen den Klassen `Heizung` und `Thermostat` muss dann ein Adapter vermitteln:

```
class HeizungThermostatAdapter : IThermostatListener {  
  
    private readonly Heizung heizung;  
    public HeizungThermostatAdapter(Heizung heizung) {  
        this.heizung = heizung;  
    }  
  
    public void MinAlarm() {  
        this.heizung.BrennerEin();  
    }  
    public void MaxAlarm() {  
        this.heizung.BrennerAus();  
    }  
}
```

Es ist nun diese Adapter-Klasse, welche das Interface `ThermostatListener` implementiert – ein entsprechendes Adapter-Objekt kann somit an den Konstruktor des `Thermostats` übergeben werden. Dem Konstruktor der Adapter-Klasse selbst muss eine `Heizung` übergeben werden.

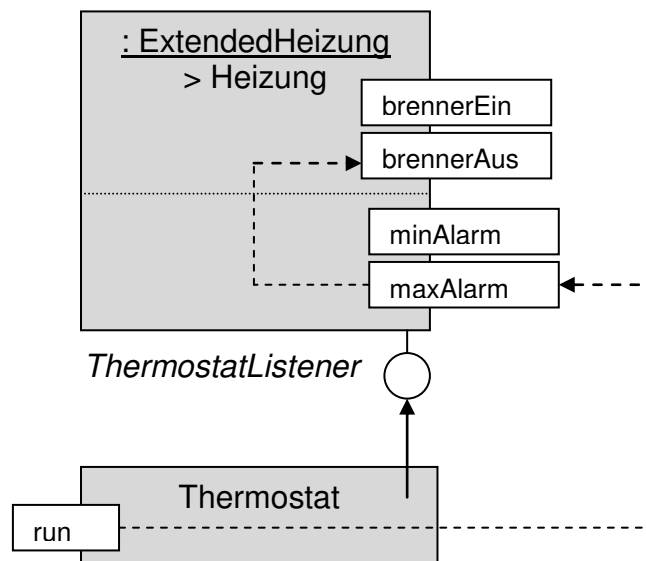
Und hier die Nutzung der drei Klassen:

```
package appl;  
  
import heinz.Heizung;  
import theo.Thermostat;  
  
public class Application {  
  
    public static void main(String[] args) {  
        final Heizung h = new Heizung();  
        final HeizungThermostatAdapter a = new  
HeizungThermostatAdapter(h);  
        final Thermostat t = new Thermostat(a);  
        t.run();  
    }  
}
```

Resultat: Ein `Thermostat` kann auch dann eine `Heizung` (einen Kuehlschrank) steuern, wenn dies weder von den `Thermostatbauern` noch von den `Heizungsbauern` (den `Kühlschrankbauern`) vorgesehen war. Ein Adapter ermöglicht also das Zusammenspiel von Objekten auch dann, wenn dieses Zusammenspiel nicht von vornherein vorgesehen war.

Die Adapter-Klasse ist eine Klasse, die für den aktuellen Zweck maßgeschneidert ist; sie ist also nicht wiederverwendbar. (Und für einen Kühlschrank muss es natürlich eine andere maßgeschneiderte Adapter-Klasse geben.)

3.1.3 Klassen-Adapter



Statt eines Objektadapters (wie im letzten Beispiel) könnte auch ein Klassenadapter verwendet werden. Ein Klassenadapter kommt ohne zusätzliches Objekt aus; stattdessen wird die Zielklasse der Adaption (hier: die `Heizung`) als Basisklasse für eine Ableitung verwendet, welche ihrerseits das verlangte Interface implementiert:

```

package appl;

import heinz.Heizung;
import theo.ThermostatListener;

public class ExtendedHeizung
    extends Heizung implements ThermostatListener {
    @Override
    public void minAlarm() {
        this.brennerEin();
    }
    @Override
    public void maxAlarm() {
        this.brennerAus();
    }
}
  
```

Die Anwendung kommt dann mit nur zwei Objekten aus:

```
package appl;

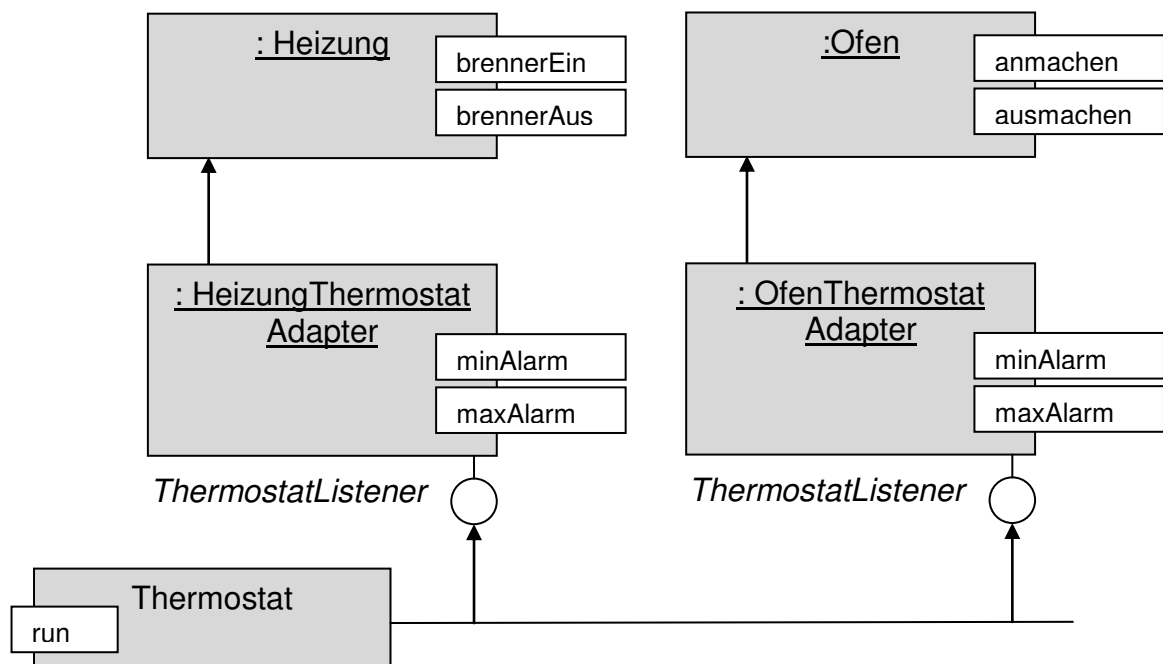
import theo.Thermostat;

public class Application {
    public static void main(String[] args) {
        final ExtendedHeizung h = new ExtendedHeizung();
        final Thermostat t = new Thermostat(h);
        t.run();
    }
}
```

I.d.R. sollte aber eine Lösung, die auf "Objektkomposition via Interfaces" beruht, einer vererbungsbasierten Lösung vorgezogen werden. Die erste Lösung ist im allgemeinen flexibler.

Denn Vererbung bedeutet, dass bereits zur Compilationszeit Teile fest zusammengeschweißt werden. Objektkomposition dagegen findet erst zur Laufzeit statt; und Elemente, die zur Laufzeit miteinander verbunden wurden, können auch zur Laufzeit wieder voneinander getrennt werden.

3.1.4 Listener-Registratur



Die Thermostat-Klasse kann dahingehend erweitert werden, dass bei einem Thermostat mehrere ThermostatListener registriert werden können:

```

package theo;

import java.util.ArrayList;
import java.util.List;

public class Thermostat {

    private final List<ThermostatListener> listeners = new ArrayList<>();

    public void addThermostatListener(ThermostatListener listener) {
        this.listeners.add(listener);
    }

    public void removeThermostatListener(ThermostatListener listener) {
        this.listeners.remove(listener);
    }

    public void run() {
        // es ist zu kalt geworden
        for (final ThermostatListener listener : this.listeners)
            listener.minAlarm();

        // es ist zu warm geworden
        for (final ThermostatListener listener : this.listeners)
            listener.maxAlarm();
    }
}
  
```

Statt wie bislang bereits dem Konstruktor des `Thermostats` einen (einen einzigen!) Listener zu übergeben, kann hier nun eine Vielzahl von Listeners mittels des wiederholten Aufrufs von `addThermostatListener` registriert werden. In der `run`-Methode werden dann alle registrierten Listener entsprechend benachrichtigt.

Hier zunächst eine weitere zu adaptierende Klasse:

```
package heinz;

public class Ofen {
    public void anmachen() {
        System.out.println("anmachen");
    }
    public void ausmachen() {
        System.out.println("ausmachen");
    }
}
```

Hier eine entsprechende Adapter-Klasse:

```
package appl;

import heinz.Ofen;
import theo.ThermostatListener;

public class OfenThermostatAdapter implements ThermostatListener {

    private final Ofen ofen;

    public OfenThermostatAdapter(Ofen ofen) {
        this.ofen = ofen;
    }

    @Override
    public void minAlarm() {
        this.ofen.anmachen();
    }

    @Override
    public void maxAlarm() {
        this.ofen.ausmachen();
    }
}
```

Und hier schließlich eine Anwendung, in welcher ein `Thermostat` sowohl eine `Heizung` als auch einen `Ofen` steuert:

```
package appl;

import heinz.Heizung;
import heinz.Ofen;
import theo.Thermostat;

public class Application {

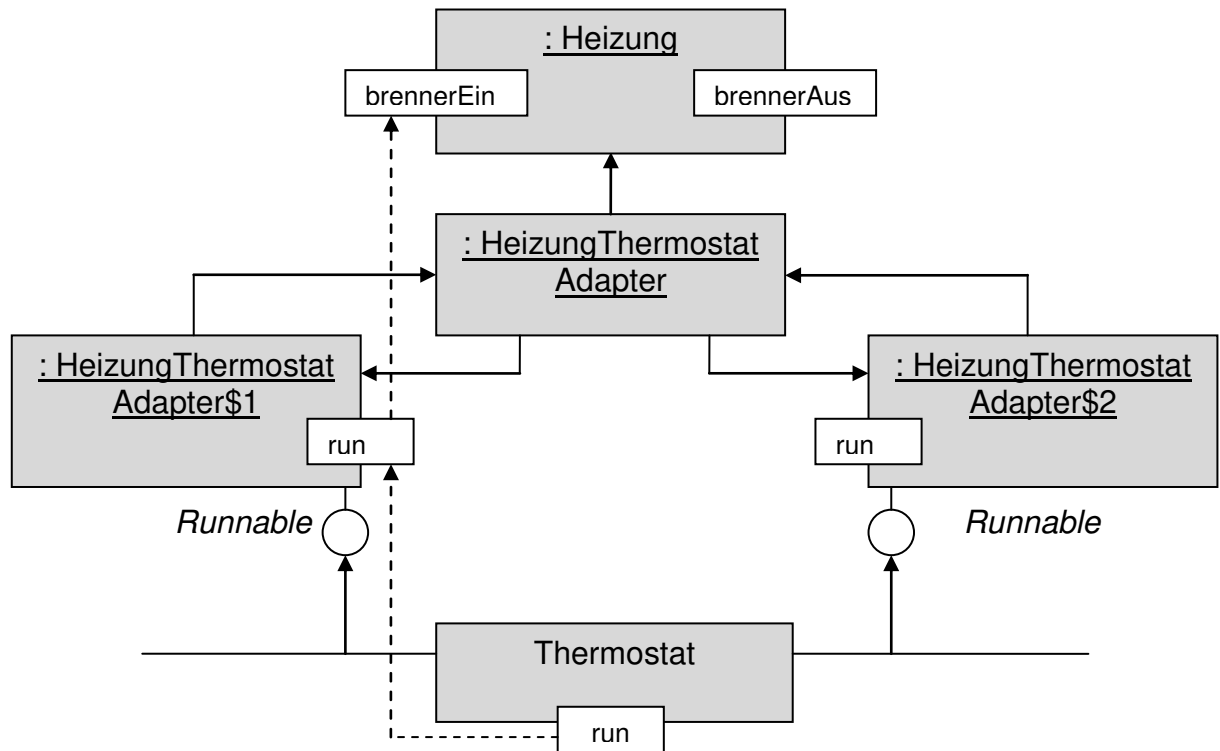
    public static void main(String[] args) {
        final Heizung h = new Heizung();
        final Ofen o = new Ofen();
        final Thermostat t = new Thermostat();
        t.addThermostatListener(new HeizungThermostatAdapter(h));
        t.addThermostatListener(new OfenThermostatAdapter(o));
        t.run();
    }
}
```

Die Ausgabe des obigen Programms:

```
Brenner aus
ausmachen
Brenner ein
anmachen
```


3.1.5 Runnable

Wir können eine `Thermostat`-Klassen definieren, welches ganz ohne das bislang verwendete Interface `ThermostatListener` auskommt – eine Klasse, welche einfach das Standard-Interface `Runnable` nutzt:



```
package theo;

import java.util.ArrayList;
import java.util.List;

public class Thermostat {

    private final List<Runnable> minListeners = new ArrayList<>();
    private final List<Runnable> maxListeners = new ArrayList<>();

    public void addMinListener(Runnable listener) {
        this.minListeners.add(listener);
    }

    public void removeMinListener(Runnable listener) {
        this.minListeners.remove(listener);
    }

    public void addMaxListener(Runnable listener) {
        this.maxListeners.add(listener);
    }

    public void removeMaxListener(Runnable listener) {
        this.maxListeners.remove(listener);
    }
}
```

```
public void run() {  
    // es ist zu kalt geworden  
    for (final Runnable listener : this.minListeners)  
        listener.run();  
  
    // es ist zu warm geworden  
    for (final Runnable listener : this.maxListeners)  
        listener.run();  
}
```

Ein Thermostat hat zwei Listen – Listen, in welchen jeweils `Runnable`s registriert werden können: `minListeners` und `maxListener`. Ist es zu kalt geworden, wird die `run`-Methode aller in der `minListeners`-Liste registrierten `Runnable`s aufgerufen; ist es zu warm geworden, wird die `run`-Methode aller in der `maxListeners`-Liste eingetragenen `Runnable`s aufgerufen.

Dann müssen natürlich auch die Adapter-Klassen anpassen. Hier der Adapter für die Heizung:

```
package appl;  
  
import heinz.Heizung;  
  
public class HeizungThermostatAdapter {  
  
    private final Heizung heizung;  
  
    public HeizungThermostatAdapter(Heizung heizung) {  
        this.heizung = heizung;  
    }  
  
    public final Runnable maxListener = new Runnable() {  
        @Override  
        public void run() {  
            HeizungThermostatAdapter.this.heizung.brennerAus();  
        }  
    };  
  
    public final Runnable minListener = new Runnable() {  
        @Override  
        public void run() {  
            HeizungThermostatAdapter.this.heizung.brennerEin();  
        }  
    };  
}
```

Ein `HeizungThermostatAdapter` erzeugt zwei Objekte einer jeweils das `Runnable`-Interface implementierenden anonymen Klassen – Objekte, die den öffentlichen Konstanten `maxListener` und `minListener` zugewiesen werden. Die `run`-Methode des ersten Objekts ruft die `brennerAus`-Methode der Heizung auf, die `run`-Methode des zweiten Objekts ruft die `brennerEin`-Methode der Heizung auf.

Im Hauptprogramm wird das `maxListeners-Runnable` an die `addMaxListeners-Methode` des `Thermostats` übergeben und das `minListener-Runnable` an die `addMinListeners-Methode`:

```
package appl;

import heinz.Heizung;
import theo.Thermostat;

public class Application {

    public static void main(String[] args) {
        final Heizung h = new Heizung();
        final Thermostat t = new Thermostat();
        final HeizungThermostatAdapter a =
            new HeizungThermostatAdapter(h);
        t.addMaxListener(a.maxListener);
        t.addMinListener(a.minListener);
        t.run();
    }
}
```

3.1.6 Lambdas

Mit den Mitteln von Java 8 können wir die Adapter-Klasse radikal vereinfachen. Statt wie

bei der letzten Erweiterung anonyme Klassen zu verwenden, welche das `Runnable`-Interface implementieren, verwenden wird nun Lambdas:

```
package appl;

import heinz.Heizung;

public class HeizungThermostatAdapter {

    private final Heizung heizung;

    public HeizungThermostatAdapter(Heizung heizung) {
        this.heizung = heizung;
    }

    public final Runnable maxListener =
        () -> HeizungThermostatAdapter.this.heizung.brennerAus();

    public final Runnable minListener =
        () -> HeizungThermostatAdapter.this.heizung.brennerEin();
}
```

3.1.7 Ad-hoc-Lambdas

Schließlich können wir auf eine explizite Adapter-Klasse auch gänzlich verzichten und die Adaption direkt in der `main`-Methode erledigt:

```
package appl;

import heinz.Heizung;
import theo.Thermostat;

public class Application {

    public static void main(String[] args) {
        final Heizung h = new Heizung();
        final Thermostat t = new Thermostat();
        t.addMaxListener(() -> h.brennerAus());
        t.addMinListener(() -> h.brennerEin());
        t.run();
    }
}
```

Aufgaben

Beim Aufruf einer Methode eines wie auch immer gearteten Listeners wird i.d.R. ein Event-Objekt übergeben, welches das gemeldete Ereignis näher beschreibt (z. B. die Quelle des Ereignisses, die Position der Maus (beim `MouseEvent`), die betätigte Taste (beim `KeyEvent`) etc. Im Falle des `Thermostats` könnte im Event-Objekt z. B. die aktuelle Temperatur eingetragen sein.

Erweitern Sie die letzte Lösung derart, dass den aufgerufenen Listener-Methoden ein solcher Event übergeben wird. Ersetzen Sie dabei das `Runnable`-Interface (dessen `run`-Methode bekanntlich eine leere Parameterliste besitzt) durch das in Java 8 eingeführte `Consumer`-Interface (dessen `accept`-Methode mit einem Parameter aufgerufen wird).

3.1.8 Swing: ActionListener

Im folgenden werden Adapter gezeigt, welche das Interface ActionListener implementieren. Solche Adapter vermitteln z.B. zwischen JButton-Objekten und einem Anwendungs-Objekt, welches die eigentliche Funktionalität enthält.

Es werden sechs Varianten der Implementierung von ActionListener gezeigt.

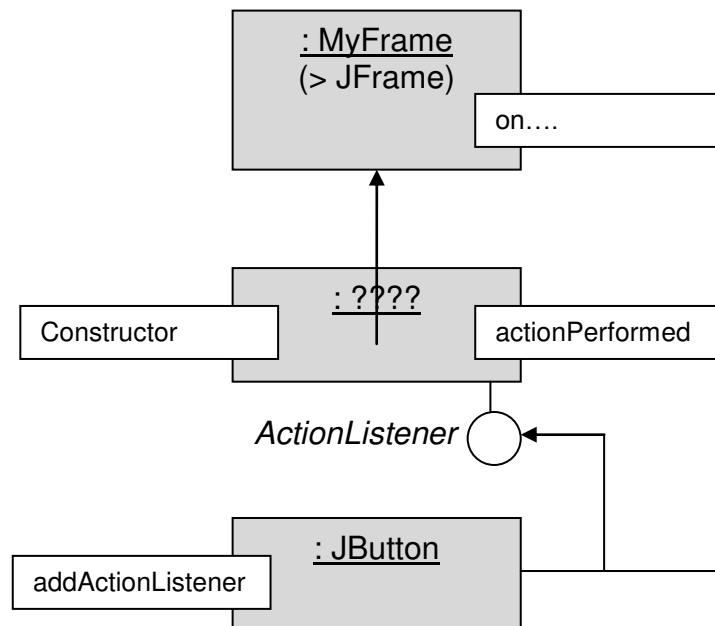
Die Anwendung präsentiert sich wie folgt:



Die Betätigung eines der sechs Buttons führt jeweils dazu, dass eben dieser Button disabled wird. (Die Buttons "One" und "Two" sind also offenbar bereits betätigt worden).

Die Betätigung jedes Buttons führt dazu, dass jeweils eine spezifische Verarbeitungsmethode aufgerufen wird: `onOne`, `onTwo`, `onThree` etc. Innerhalb dieser Methoden wird der jeweilige Button disabled.

Ein kleines Objektdiagramm (welches allen sechs Varianten explizit oder aber implizit zugrunde liegt):



Zunächst eine "globale" Adapter-Klasse:

```
package gui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ButtonOneAdapter implements ActionListener {
    private final MyFrame frame;

    public ButtonOneAdapter(MyFrame frame) {
        this.frame = frame;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        this.frame.onOne();
    }
}
```

Alle anderen Adapter-Klassen sind im Kontext der Klasse `MyFrame` implementiert – in folgenden Varianten:

- statische Member-Klasse
- nicht statische Member-Klasse
- lokale Klasse
- anonyme Klasse
- Lambda-Ausdruck

```
package gui;

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

public class MyFrame extends JFrame {

    private static class ButtonTwoAdapter implements ActionListener {
        private final MyFrame frame;

        public ButtonTwoAdapter(MyFrame frame) {
            this.frame = frame;
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            this.frame.onTwo();
        }
    }

    private class ButtonThreeAdapter implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
```

```
        MyFrame.this.onThree();
    }
}

private final JButton buttonOne = new JButton("One");
private final JButton buttonTwo = new JButton("Two");
private final JButton buttonThree = new JButton("Three");
private final JButton buttonFour = new JButton("Four");
private final JButton buttonFive = new JButton("Five");
private final JButton buttonSix = new JButton("Six");

public MyFrame() {
    this.setLayout(new FlowLayout());

    this.add(this.buttonOne);
    this.add(this.buttonTwo);
    this.add(this.buttonThree);
    this.add(this.buttonFour);
    this.add(this.buttonFive);
    this.add(this.buttonSix);

    this.registerListeners();
    this.setBounds(100, 100, 400, 100);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setVisible(true);
}

private void registerListeners() {
    // global class
    this.buttonOne.addActionListener(new ButtonOneAdapter(this));

    // static member class
    this.buttonTwo.addActionListener(new ButtonTwoAdapter(this));

    // non-static member class
    this.buttonThree.addActionListener(new ButtonThreeAdapter());

    // local class
    class ButtonFourAdapter implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            MyFrame.this.onFour();
        }
    }
    this.buttonFour.addActionListener(new ButtonFourAdapter());

    // anonymous class
    this.buttonFive.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            MyFrame.this.onFive();
        }
    });

    // Lambda
    this.buttonSix.addActionListener(e -> this.onSix());
}

public void onOne() {
```



```
        this.buttonOne.setEnabled(false);
    }

    private void onTwo() {
        this.buttonTwo.setEnabled(false);
    }

    private void onThree() {
        this.buttonThree.setEnabled(false);
    }

    private void onFour() {
        this.buttonFour.setEnabled(false);
    }

    private void onFive() {
        this.buttonFive.setEnabled(false);
    }

    private void onSix() {
        this.buttonSix.setEnabled(false);
    }
}
```

Das Hauptprogramm beschränkt sich auf die Erzeugung eines `MyFrame`-Objekts:

```
package appl;

import gui.MyFrame;

public class Application {
    public static void main(String[] args) {
        new MyFrame();
    }
}
```

3.1.9 Swing: TreeModel

Im folgenden bauen wir eine Adapter-Klasse, welche das Swing-Interface `TreeModel` implementiert. Der Adapter bringt einen `JTree` mit einer baumförmigen Datenstruktur zusammen, welche ein Unternehmen abbildet (eine `Company` hat `Departments`, ein `Department` hat `Employees`):



Hier die Domain-Klassen (die von Swing nichts wissen!):

```
package domain;

public class Employee {
    public final int number;
    public final String name;
    public Employee(int number, String name) {
        this.number = number;
        this.name = name;
    }
    @Override
    public String toString() {
        return this.number + " " + this.name;
    }
}
```

```
package domain;
// ...
public class Department {
    public final int number;
    public final String name;
    public final List<Employee> employees = new ArrayList<>();
    public Department(int number, String name) {
        this.number = number;
        this.name = name;
    }
    @Override
    public String toString() {
        return this.number + " " + this.name;
    }
}
```

```
package domain;
// ...
public class Company {
    public final int number;
    public final String name;
    public final List<Department> departments = new ArrayList<>();
    public Company(int number, String name) {
        this.number = number;
        this.name = name;
    }
    @Override
    public String toString() {
        return this.number + " " + this.name;
    }

    public static final Company example = new Company(1, "Siemens");

    static {
        final Department d1 = new Department(10, "Einkauf");
        final Department d2 = new Department(20, "Verkauf");
        final Department d3 = new Department(30, "Entwicklung");

        final Employee e1 = new Employee(100, "Meier");
        final Employee e2 = new Employee(200, "Mueller");
        final Employee e3 = new Employee(300, "Franke");
        final Employee e4 = new Employee(400, "Schulte");
        final Employee e5 = new Employee(500, "Schmidt");
        final Employee e6 = new Employee(700, "Klute");

        example.departments.add(d1);
        example.departments.add(d2);
        example.departments.add(d3);

        d1.employees.add(e1);
        d1.employees.add(e2);
        d2.employees.add(e3);
        d2.employees.add(e4);
        d2.employees.add(e5);
        d3.employees.add(e6);
    }
}
```

Bei der Konstruktion dieser Klassen wurde auf Datenkapselung der Einfachheit halber verzichtet...

Im statischen Block der Klasse `Company` wird ein beispielhaftes Unternehmen aufgebaut (`example`).

Um einen Baum (beliebiger Struktur) in einem JTree anzeigen zu können, müssen wir das Interface `TreeModel` implementieren – der JTree bezieht über dieses Interface die von ihm darzustellenden Daten):

```
package gui;

import javax.swing.event.TreeModelListener;
import javax.swing.tree.TreeModel;
import javax.swing.tree.TreePath;

import domain.Company;
import domain.Department;
import domain.Employee;

public class CompanyTreeModel implements TreeModel {
    private final Company root;

    public CompanyTreeModel(Company root) {
        this.root = root;
    }

    @Override
    public Object getRoot() {
        return this.root;
    }

    @Override
    public int getChildCount(Object parent) {
        if (parent instanceof Company)
            return ((Company) parent).departments.size();
        if (parent instanceof Department)
            return ((Department) parent).employees.size();
        return 0;
    }

    @Override
    public Object getChild(Object parent, int index) {
        if (parent instanceof Company)
            return ((Company) parent).departments.get(index);
        if (parent instanceof Department)
            return ((Department) parent).employees.get(index);
        throw new RuntimeException();
    }

    @Override
    public boolean isLeaf(Object node) {
        return node instanceof Employee;
    }

    @Override
    public int getIndexOfChild(Object parent, Object child) {
        for (int i = 0; i < this.getChildCount(parent); i++)
            if (this.getChild(parent, i) == child)
                return i;
        return -1;
    }

    @Override
    public void valueForPathChanged(TreePath path, Object newValue) {
```

```
    }

    @Override
    public void addTreeModelListener(TreeModelListener l) {
    }

    @Override
    public void removeTreeModelListener(TreeModelListener l) {
    }
}
```

Über den Aufruf von `setModel` wird der `JTree` mit einem `TreeModel` verbunden:

```
package gui;

import java.awt.Dimension;
import java.awt.FlowLayout;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;

import domain.Company;

public class MyFrame extends JFrame {

    private final JTree tree = new JTree();
    private final JScrollPane scrollPane = new JScrollPane(this.tree);

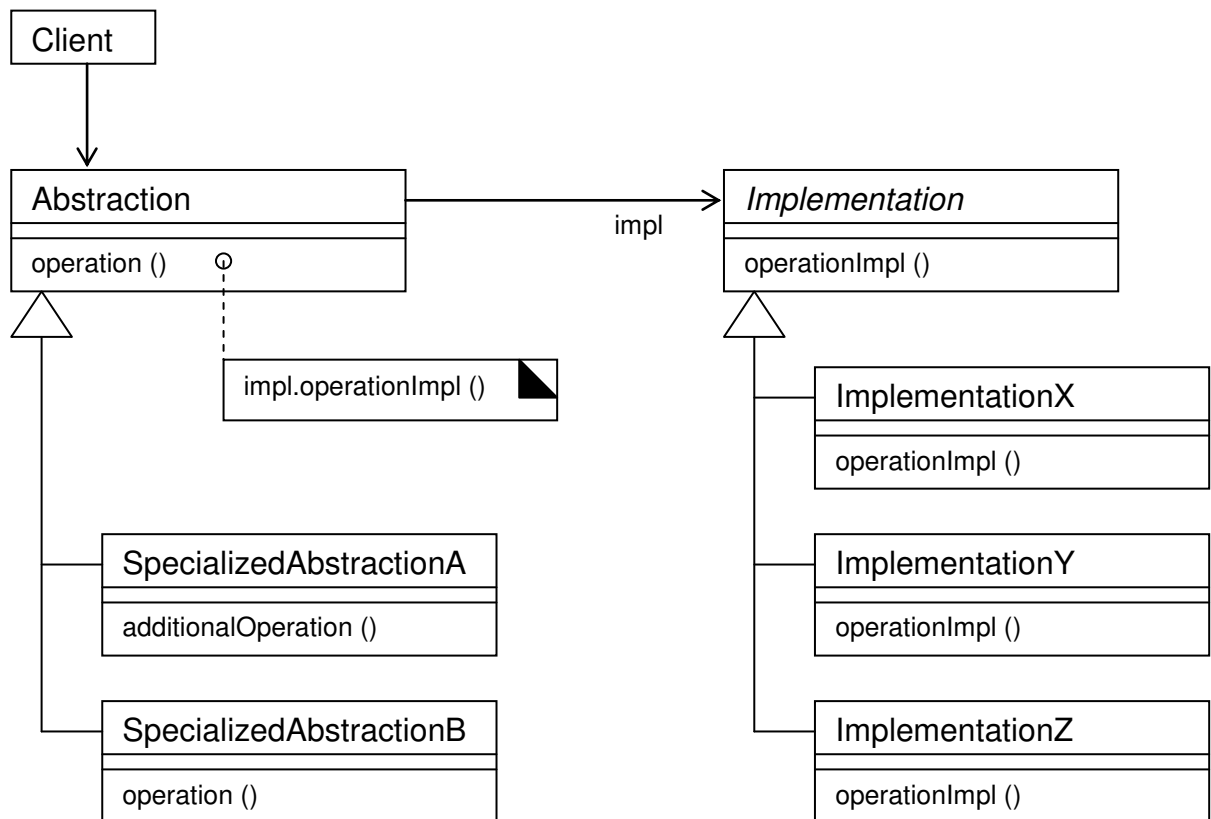
    public MyFrame() {
        this.setLayout(new FlowLayout());
        this.scrollPane.setPreferredSize(new Dimension(400, 400));
        this.add(this.scrollPane);

        this.tree.setModel(new CompanyTreeModel(Company.example));

        this.setLocation(100, 100);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }
}
```

3.2 Bridge

"Entkopple eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können." (Gamma, 165)



3.2.1 Problem

Properties (key-value-Paare) können in einer einfacher Textdatei oder in einer XML-Datei beschrieben werden.

Hier eine Textdatei ("properties.properties"):

```
colors = red blue green
fontname = arial
fontsize = 12
```

Und hier eine Properties-Beschreibung in XML-Form ("properties.xml"):

```
<?xml version='1.0'?>
<!DOCTYPE properties SYSTEM "properties.dtd">

<properties>
  <property>
    <key>colors</key>
    <value>yellow black orange</value>
  </property>
  <property>
    <key>fontsize</key>
    <value>16</value>
  </property>
  <property>
    <key>fontname</key>
    <value>Courier</value>
  </property>
</properties>
```

Dem obigen Dokument liegt folgende DTD zugrunde ("properties.dtd"):

```
<!ELEMENT properties (property*)>
<!ELEMENT property (key, value)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT value (#PCDATA)>
```

Man könnte dann zwei Klassen schreiben, welche für das Einlesen solcher Properties zuständig sind. Das Einlesen wird im Konstruktor der beiden Klassen geschehen. Anschließend muss es möglich sein, aufgrund eines gegebenen Schlüssels den entsprechenden Wert zu ermitteln.

Es wird dann sinnvoll sein, diese Funktionalität der beiden Klassen zunächst in einem gemeinsamen Interface zu spezifizieren:

```
package util;

public interface Properties {
    public abstract String get(String key);
}
```

Hier die Klasse `TextProperties`:

```
package util;
// ...
public class TextProperties implements Properties {

    private final java.util.Properties properties =
        new java.util.Properties();

    public TextProperties(String filename) {
        try {
            this.properties.load(new FileInputStream(filename));
        }
        catch (final IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public String get(String key) {
        return this.properties.getProperty(key);
    }
}
```

Die Klasse benutzt intern ein `java.util.Properties`-Objekt zum Einlesen der Daten.

Und hier die Klasse `XmlProperties`:

```
package util;
// ...
import util.xml.XMLScanner;

public class XMLProperties implements Properties {

    private final Map<String, String> map = new HashMap<>();

    public XMLProperties(String filename) {
        try {
            final XMLScanner scanner =
                new XMLScanner(new FileInputStream(filename));
            scanner.start("properties");
            while(scanner.isStart("property")) {
                scanner.start("property");
                final String key = scanner.startTextEnd("key");
                final String value = scanner.startTextEnd("value");
                this.map.put(key, value);
                scanner.end("property");
            }
            scanner.end("properties");
        }
        catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public String get(String key) {
        return this.map.get(key);
    }
}
```


Die Klasse benutzt zum Lesen der XML-Datei die Klasse `XMLScanner` (aus dem `common-util`-Package).

Angenommen nun, man möchte in einigen Fällen zusätzliche Funktionalitäten nutzen. Einerseits möchte man numerische Werte statt mittels der `get`-Methode (der ja immer `String` liefert) mittels einer Methode `getInt` ermitteln (`getDouble`, ...); andererseits möchte man direkt indiziert auf "mehrwertige" Properties zugreifen können (siehe den Property-Wert für `"color"`!).

Dann empfiehlt sich die Konstruktion eines erweiterten Interfaces:

```
package util;

public interface ExtendedProperties extends Properties {
    public abstract String[] getValues(String key);
    public abstract int getInt(String key);
    // ...
}
```

Leider muss dieses Interface nun in zwei zusätzlichen Klassen implementiert werden (immer auf dieselbe Weise!) – sofern man sich weiterhin auf die Text- als auch auf die XML-Form von Properties beziehen will.

Für die Text-Form wird folgende von `TextProperties` abgeleitete Klasse definiert:

```
package util;

public class ExtendedTextProperties
    extends TextProperties implements ExtendedProperties {
    public ExtendedTextProperties(String filename) {
        super(filename);
    }
    @Override
    public String[] getValues(String key) {
        final String value = this.get(key);
        return value == null ? new String[0] : value.split(" ");
    }
    @Override
    public int getInt(String key) {
        return Integer.parseInt(this.get(key));
    }
    // ...
}
```

Und für die XML-Form wird von der Klasse `XmlProperties` abgeleitet:

```
package util;

public class ExtendedXMLProperties
    extends XMLProperties implements ExtendedProperties {
    public ExtendedXMLProperties(String filename) {
        super(filename);
    }
    @Override
    public String[] getValues(String key) {
        final String value = this.get(key);
        return value == null ? new String[0] : value.split(" ");
    }
    @Override
    public int getInt(String key) {
        return Integer.parseInt(this.get(key));
    }
    // ...
}
```

Man beachte, dass die beiden Implementierung im Prinzip identisch sind...

Hier eine kleine Anwendung der Klassen:

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        final Properties props1 =
            new TextProperties("src/properties.txt");
        print(props1);

        final Properties props2 =
            new XMLProperties("src/properties.xml");
        print(props2);

        final ExtendedProperties props3 =
            new ExtendedTextProperties("src/properties.txt");
        print(props3);

        final ExtendedProperties props4 =
            new ExtendedXMLProperties("src/properties.xml");
        print(props4);
    }

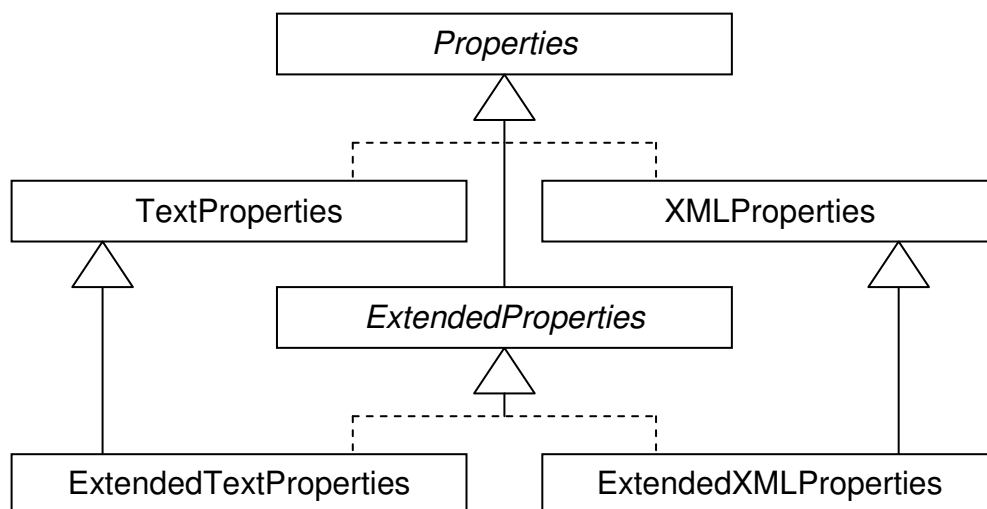
    static void print(Properties props) {
        System.out.println("colors = " + props.get("colors"));
        System.out.println("fontsize = " + props.get("fontsize"));
        System.out.println("fontname = " + props.get("fontname"));
    }

    static void print(ExtendedProperties props) {
        System.out.println("colors = ");
        for (final String value : props.getValues("colors"))
            System.out.println("\t" + value);
        System.out.println("fontsize = " + props.getInt("fontsize"));
        System.out.println("fontname = " + props.get("fontname"));
    }
}
```

Hie schließlich die Ausgaben:

```
colors = red blue green
fontsize = 12
fontname = arial
colors = yellow black orange
fontsize = 16
fontname = Courier
colors =
    red
    blue
    green
fontsize = 12
fontname = arial
colors =
    yellow
    black
    orange
fontsize = 16
fontname = Courier
```

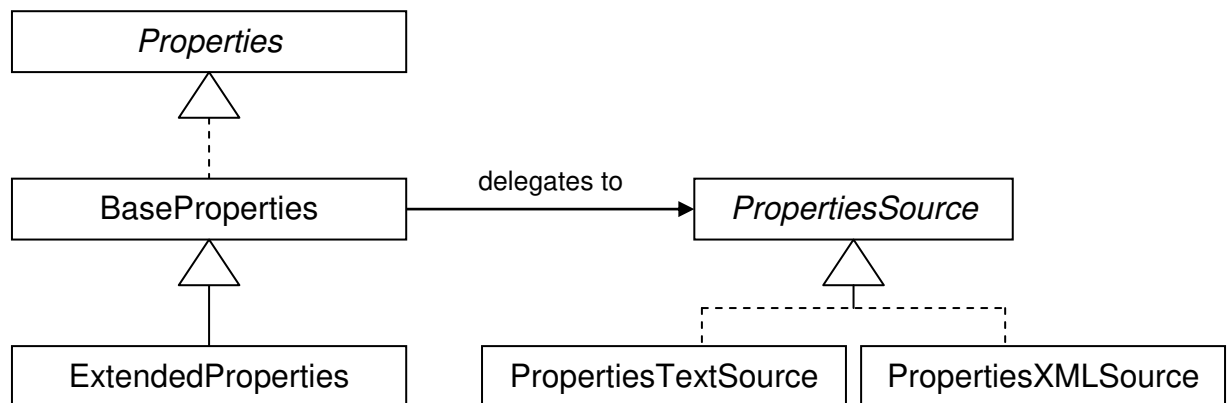
Das Klassendiagramm für der obigen Implementierung:



Es wäre besser, die "logische" Nutzung von *Properties* von der "Quelle" der *Properties* zu trennen – um auf diese Weise dann die Redundanz in der Definition der *Extended*-Klassen vermeiden zu können.

3.2.2 Lösung

Die Klassenhierarchie wird in zwei Hierarchien aufgesplittet:



```
package util;

public interface Properties {
    public abstract String get(String key);
}
```

```
package util;

public interface PropertiesSource {
    public abstract String get(String key);
}
```

Die Klasse `TextProperties` und `XMLProperties` werden umgetauft: sie bekommen die Namen `PropertiesTextSource` `PropertiesXMLSource` und implementieren das Interface `PropertiesSource`:

```
package util;
// ...
public class PropertiesTextSource implements PropertiesSource {
    private final java.util.Properties properties =
        new java.util.Properties();
    public PropertiesTextSource(String filename) { ... }
    @Override
    public String get(String key) {
        return this.properties.getProperty(key);
    }
}
```

```
package util;
// ...
public class PropertiesXMLSource implements PropertiesSource {
    private final Map<String, String> map = new HashMap<>();
    public PropertiesXMLSource(String filename) { ... }
    @Override
    public String get(String key) {
        return this.map.get(key);
    }
}
```

Zum einfachen Zugriff auf Properties wird die Klasse `BaseProperties` eingeführt – eine Klasse, die das Interface `Property` implementiert. Der Konstruktor der Klasse verlangt ein Objekt, dessen Klasse das Interface `PropertySource` implementiert. Die `get`-Methode von `BaseProperty` kann dann einfach an die `get`-Methode dieses `PropertySource`-Objekts delegieren.

```
package util;

public class BaseProperties implements Properties {
    private final PropertiesSource source;
    public BaseProperties(PropertiesSource source) {
        this.source = source;
    }
    @Override
    public String get(String key) {
        return this.source.get(key);
    }
}
```

Man beachte, dass ein `BaseProperties`-Objekt nun sowohl mit einem Objekt vom Typ `PropertiesTextSource` als auch mit einem Objekt des Typs `PropertiesXMLSource` zusammenarbeiten kann. Die "Quelle" der Daten ist für den "Zugriff" auf die Daten transparent geworden.

Von `BaseProperties` kann nun `ExtendedProperties` abgeleitet werden:

```
package util;

public class ExtendedProperties extends BaseProperties {
    public ExtendedProperties(PropertiesSource source) {
        super(source);
    }
    public String[] getValues(String key) {
        final String value = this.get(key);
        return value == null ? new String[0] : value.split(" ");
    }
    public int getInt(String key) {
        return Integer.parseInt(this.get(key));
    }
    // ...
}
```

Es gibt nun also eine Abstraktion für die Implementierung ("Quelle") – nämlich `PropertiesSource`. Es existiert eine zweite Abstraktion für die "Nutzung" – das Interface `Properties`. Aber beide Abstraktionen sind nun unabhängig und können als Grundlage für jeweils eigene Erweiterungen benutzt werden. Und die Klassen, die für die "Nutzung" vorgesehen sind, delegieren einfach an eine der "Source"-Klassen.

Hier eine Anwendung der Klassen:

```
package appl;

import util.BaseProperties;
import util.ExtendedProperties;
import util.PropertiesTextSource;
import util.PropertiesXMLSource;

public class Application {
    public static void main(String[] args) {

        final BaseProperties props1 = new BaseProperties(
            new PropertiesTextSource("src/properties.txt"));
        print(props1);

        final BaseProperties props2 = new BaseProperties(
            new PropertiesXMLSource("src/properties.xml"));
        print(props2);

        final ExtendedProperties props3 = new ExtendedProperties(
            new PropertiesTextSource("src/properties.txt"));
        print(props3);

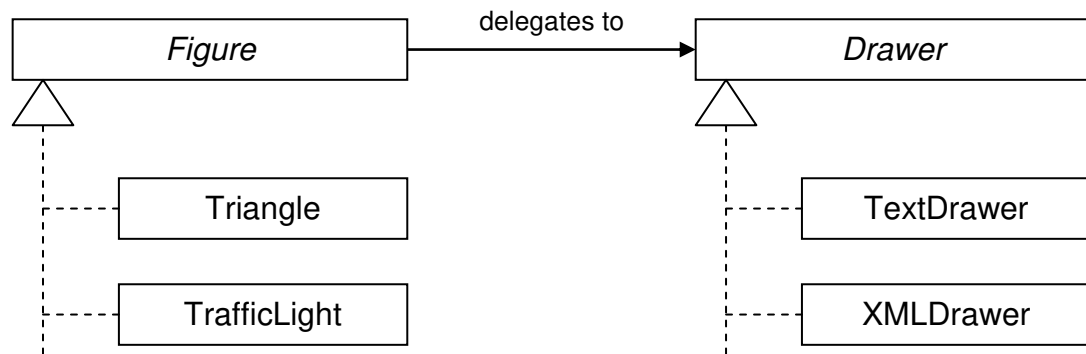
        final ExtendedProperties props4 = new ExtendedProperties(
            new PropertiesXMLSource("src/properties.xml"));
        print(props4);
    }

    static void print(BaseProperties props) {
        System.out.println("colors = " + props.get("colors"));
        System.out.println("fontsize = " + props.get("fontsize"));
        System.out.println("fontname = " + props.get("fontname"));
    }

    static void print(ExtendedProperties props) {
        System.out.println("colors = ");
        for (final String value : props.getValues("colors"))
            System.out.println("\t" + value);
        System.out.println("fontsize = " + props.getInt("fontsize"));
        System.out.println("fontname = " + props.get("fontname"));
    }
}
```

3.2.3 Figures und Drawers

Figures sollen gezeichnet werden können – mittels `Drawers`.



Das Interface `Drawer`:

```
package util;

public interface Drawer {
    public abstract void drawLine(int x0, int y0, int x1, int y1);
    public abstract void drawOval(int x, int y, int width, int
height);
}
```

Ein `Drawer` muss eine Linie und ein Oval zeichnen können.

Das Interface `Figure`:

```
package util;

public interface Figure {
    public abstract void draw(Drawer drawer);
}
```

Eine `Figur` muss sich zeichnen können – mittels eines `Drawers`.

Das Interface `Drawer` könnte man für den Bildschirm oder für den Drucker implementieren. Der Einfachheit halber zeichnen wir in eine Textdatei bzw. eine XML-Datei:

```
package drawers;

import util.Drawer;

public class TextDrawer implements Drawer {
    @Override
    public void drawLine(int x0, int y0, int x1, int y1) {
        System.out.println(
            "Line(" + x0 + ", " + y0 + ", " + x1 + " " + y1 + ")");
    }
    @Override
    public void drawOval(int x, int y, int width, int height) {
        System.out.println(
            "Oval(" + x + ", " + y + ", " + width + " " + height + ")");
    }
}
```

```
package drawers;

import util.Drawer;

public class XMLDrawer implements Drawer {
    @Override
    public void drawLine(int x0, int y0, int x1, int y1) {
        System.out.printf(
            "<line x0='%d' y0='%d' x1='%d', y1='%d' />\n",
            x0, y0, x1, y1);
    }
    @Override
    public void drawOval(int x, int y, int width, int height) {
        System.out.printf(
            "<oval x='%d' y='%d' width='%d', height='%d' />\n",
            x, y, width, height);
    }
}
```

Wir können viele Figuren erfinden – zwei sollen reichen: ein Dreieck und eine Ampel.

Um ein Dreieck zu zeichnen, müssen wir drei Linien zeichnen:

```
package figures;

import util.Drawer;
import util.Figure;

public class Triangle implements Figure {

    public final int x;
    public final int y;
    public final int width;
    public final int height;

    public Triangle(int x, int y, int width, int height) { ... }

    @Override
    public void draw(Drawer drawer) {
        drawer.drawLine(this.x, this.y + this.height,
            this.x + this.width, this.y + this.height);
        drawer.drawLine(...);
        drawer.drawLine(...);
    }
}
```

Eine Ampel besteht aus drei Kreisen und vier Linien:

```
package figures;

import util.Drawer;
import util.Figure;

public class TrafficLight implements Figure {

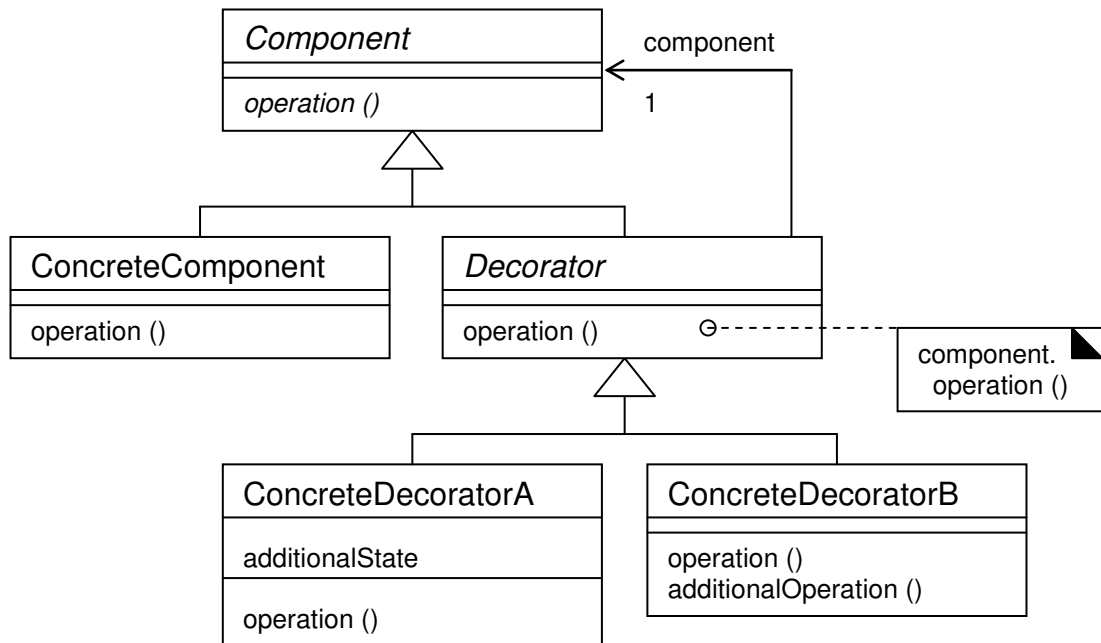
    public final int x;
    public final int y;
    public final int radius;

    public TrafficLight(int x, int y, int radius) { ... }

    @Override
    public void draw(Drawer drawer) {
        drawer.drawLine(...);
        drawer.drawLine(...);
        drawer.drawLine(...);
        drawer.drawLine(...);
        drawer.drawOval(...);
        drawer.drawOval(...);
        drawer.drawOval(...);
    }
}
```

3.3 Decorator

"Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern." (Gamma, 177)

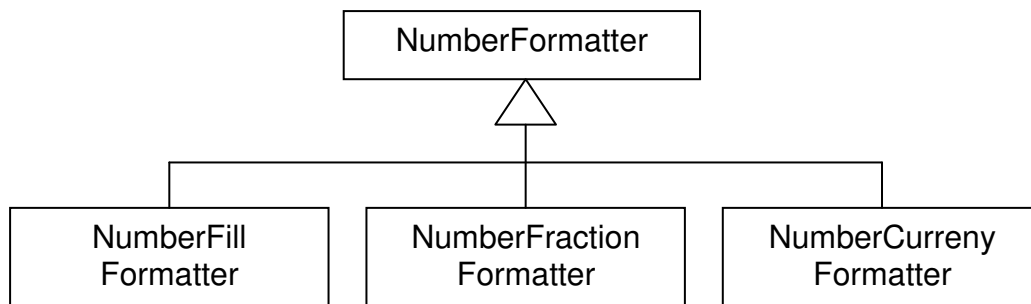


3.3.1 Problem

Zum Zwecke der Formatierung von `BigInteger`-Werten sollen `NumberFormatter` verwendet werden. `NumberFormatter` ist eine Basis-klass. Für verschiedene Formatierungsvarianten gibt's verschiedene abgeleitet Klassen:

- Um führende Nullen oder führende Blanks einzufügen (damit sich eine festgelegte Länge ergibt), gibt's die Klasse `NumberFillFormatter`
- Um Nachkomma-Stellen aufzufüllen bzw. abzuschneiden, gibt's die Klasse `NumberFractionFormatter`
- Um Währungssymbole kümmert sich die Klasse `NumberCurrencyFormatter`.

Hier das Klassendiagramm:



Die Klasse `NumberFormatter` ist trivial:

```
package util;
// ...
public class NumberFormatter {
    public String format(BigDecimal value) {
        return value.toString();
    }
}
```

Alle abgeleiteten Klassen überschreiben die `format`-Methode – und zwar derart, dass am Anfang der jeweils via `super` die `format`-Methode dieser Basisklasse aufgerufen wird.

Die Klasse NumberFillFormattter:

```
package util;
// ...
public class NumberFillFormatter extends NumberFormatter {
    private final int length;
    private final char fillChar;
    public NumberFillFormatter(final int length, final char fillChar) {
        this.length = length;
        this.fillChar = fillChar;
    }
    @Override
    public String format(BigDecimal value) {
        final String s = super.format(value);
        if (s.length() < this.length)
            return this.prependFillChars(s, this.length - s.length());
        return s;
    }
    private String prependFillChars(String s, int n) {
        while (n-- > 0)
            s = this.fillChar + s;
        return s;
    }
}
```

Die Klasse NumberFractionFormattter:

```
package util;
// ...
public class NumberFractionFormatter extends NumberFormatter {
    private final int fraction;
    public NumberFractionFormatter(int fraction) {
        this.fraction = fraction;
    }
    @Override
    public String format(BigDecimal value) {
        final String s = super.format(value);
        final int index = s.indexOf('.');
        if (index < 0)
            return s + "." + this.appendZeros("", this.fraction);
        final int n = s.length() - index - 1;
        if (n > this.fraction)
            return s.substring(0, s.length() - (n - this.fraction));
        return this.appendZeros(s, this.fraction - n);
    }
    private String appendZeros(String s, int n) {
        while (n-- > 0)
            s += '0';
        return s;
    }
}
```

Die Klasse `NumberCurrencyFormattter` (sie nutzt einen `Currency`-enum):

```
package util;
// ...
public class NumberCurrencyFormattter extends NumberFormatter {
    private final Currency currency;
    public NumberCurrencyFormattter(Currency currency) {
        this.currency = currency;
    }
    @Override
    public String format(BigDecimal value) {
        final String s = super.format(value);
        return s + " " + this.currency.symbol;
    }
}
```

Eine Demo-Applikation (und die jeweiligen Ausgaben):

```
static void demo1() {
    final NumberFormatter f = new NumberFractionFormatter(2);
    System.out.println(f.format(BigDecimal.valueOf(3)));
    System.out.println(f.format(BigDecimal.valueOf(3.1)));
    System.out.println(f.format(BigDecimal.valueOf(3.14)));
    System.out.println(f.format(BigDecimal.valueOf(3.141)));
}
```

3.00
3.10
3.14
3.14

```
static void demo2() {
    final NumberFormatter f = new NumberFillFormatter(10, '0');
    System.out.println(f.format(BigDecimal.valueOf(3)));
    System.out.println(f.format(BigDecimal.valueOf(3.14)));
    System.out.println();
}
```

0000000003
0000003.14

```
static void demo3() {
    final NumberFormatter f =
        new NumberCurrencyFormattter(Currency.EURO);
    System.out.println(f.format(BigDecimal.valueOf(3)));
    System.out.println(f.format(BigDecimal.valueOf(3.14)));
    System.out.println(f.format(BigDecimal.valueOf(3.141)));
    System.out.println(f.format(BigDecimal.valueOf(12345)));
    System.out.println();
}
```

3 €
3.14 €
3.141 €
12345 €

Angenommen, wir wollten auch wie folgt formatieren können:

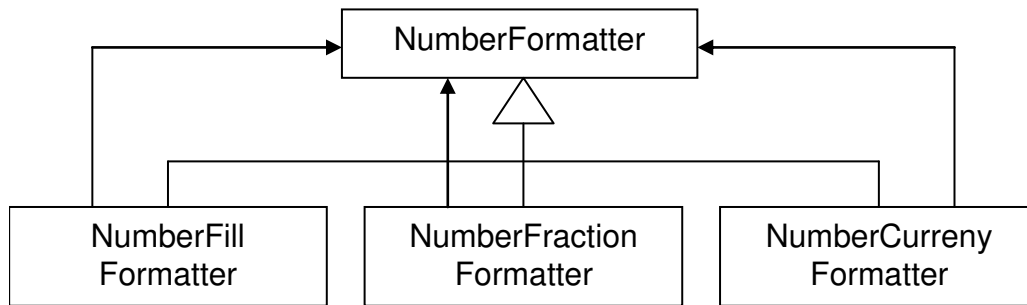
- Füllen und Nachkommastellen setzen
- Füllen und Currency setzen
- Nachkommastellen und Currency setzen
- Füllen und Nachkommastellen und Currency setzen

Wenn die obige Klassenkonstruktion beibehalten werden soll, benötigen wir dann vier neue Klassen (die allesamt eierlegende Wollmilchsäue wären)...

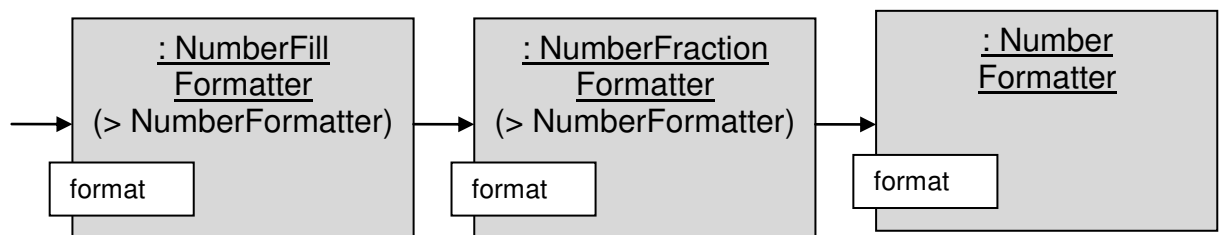
3.3.2 Lösung

Die Ableitungshierarchie bleibt bestehen. Aber jedes Objekt einer von `NumberFormatter` abgeleiteten Klasse besitzt eine Referenz auf wiederum einen `NumberFormatter` (eine Referenz, die via Konstruktor initialisiert wird).

Das Klassendiagramm:



Ein mögliches Objektdiagramm:



Diese Konstruktion erlaubt es offenbar, mehrere Formatter in Reihe zu schalten – wobei am Ende der Reihe dann ein einfaches `NumberFormatter`-Objekt stehen wird.

Die `format`-Methode in den abgeleiteten Klassen ruft nun nicht mehr via `super.format` die `format`-Methode von `NumberFormatter` auf, sondern die `format`-Methode des "rechten" Nachbars der Kette.

`NumberFillFormatter` sieht nun wie folgt aus (die beiden anderen von `NumberFormat` abgeleiteten Klassen sind nach demselben Schema geändert):

```
package util;
// ...
public class NumberFillFormatter extends NumberFormatter {
    private final NumberFormatter formatter;
    private final int length;
    private final char fillChar;

    public NumberFillFormatter(
        final int length, final char fillChar,
        final NumberFormatter formatter) {

        this.formatter = formatter;
        this.length = length;
        this.fillChar = fillChar;
    }

    @Override
    public String format(BigDecimal value) {
        final String s = this.formatter.format(value);
        if (s.length() < this.length)
            return this.prependFillChars(s, this.length - s.length());
        return s;
    }

    private String prependFillChars(String s, int n) {
        while (n-- > 0)
            s = this.fillChar + s;
        return s;
    }
}
```

Alle Formatierungs-Aspekte sind nun beliebig kombinierbar:

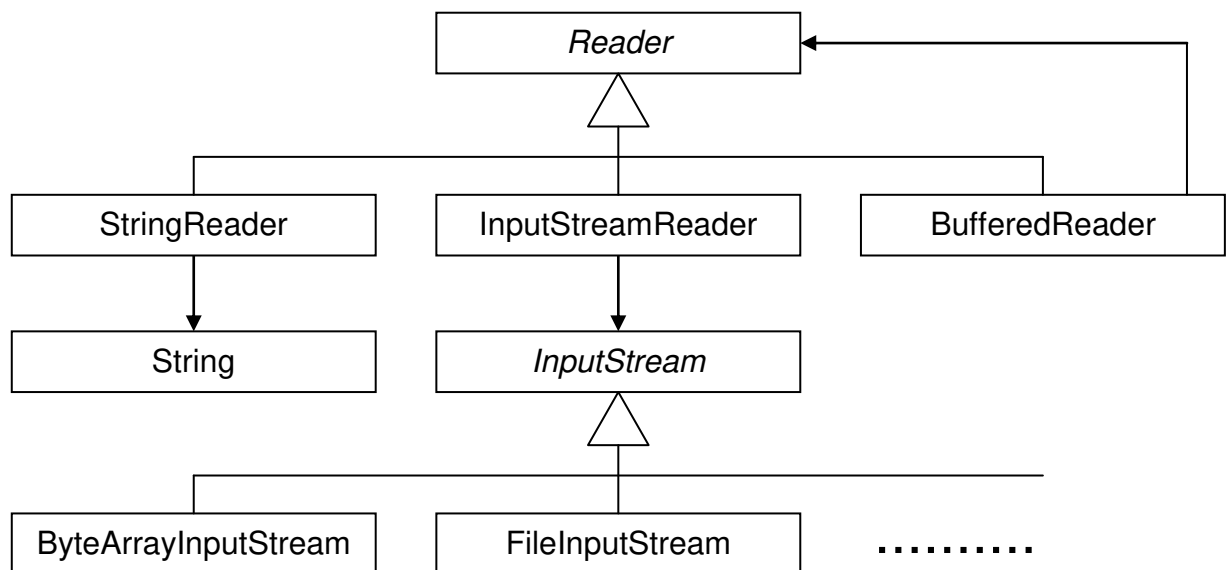
```
static void demo1() {
    final NumberFormatter f = new NumberFractionFormatter(2,
        new NumberFormatter());
    // ...
}
static void demo2() {
    final NumberFormatter f = new NumberFillFormatter(10, '0',
        new NumberFormatter());
    // ...
}
static void demo3() {
    final NumberFormatter f = new NumberFillFormatter(10, '0',
        new NumberFractionFormatter(2,
            new NumberFormatter()));
    // ...
}
static void demo4() {
    final NumberFormatter f =
        new NumberCurrencyFormatter(Currency.EURO,
            new NumberFractionFormatter(2, new
NumberFormatter()));
    // ...
}
}
```


Ein `NumberFormatter` erledigt die eigentliche "wichtige" Aufgaben – alle anderen Formattieren dekorieren das Ergebnis seiner Arbeit.

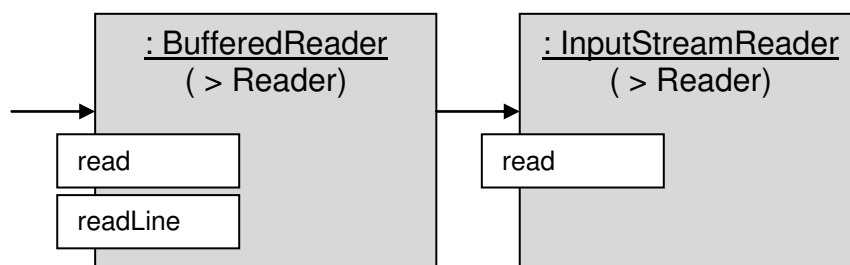
3.3.3 Reader

Wie hängen die Java-Klassen `Reader`, `InputStreamReader`, `StringReader` und `BufferedReader` zusammen – und wie kann man neue Klassen, die sich in die Architektur dieser Klassen nahtlos einfügen?

Das Klassendiagramm:



Ein mögliches Objektdiagramm:



Ein `BufferedReader` ist ein Dekorator für `Reader`. Ein `Reader` ist ein `StringReader`, ein `InputStreamReader` oder: ein `BufferedReader`. Ein `InputStreamReader` benutzt als "Quelle" eine `InputStream`; ein `InputStream` wiederum ist nur eine Basisklasse, von welcher eine Vielzahl weiterer Klassen abgeleitet sind.

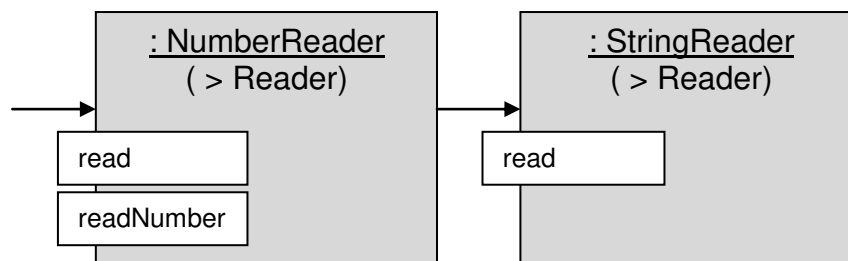
(Das Verhältnis zwischen `Reader` und `InputStream` kann übrigens als schönes Beispiel für das Bridge-Pattern angesehen werden.)

Die folgende Anwendung zeigt, wie Objekte dieser Klassen "beliebig" zusammengesetzt werden können:

```
static void demo1() throws IOException {  
    final InputStream is1 =  
        new FileInputStream("src/appl/Application.java");  
    final InputStream is2 =  
        new ByteArrayInputStream(new byte[] { 65, 66, 67 } );  
    final Reader r1 = new InputStreamReader(is1);  
    final Reader r2 = new InputStreamReader(is2);  
    final Reader r3 = new StringReader("hello");  
    final BufferedReader br1 = new BufferedReader(r1);  
    final BufferedReader br2 = new BufferedReader(r2);  
    final BufferedReader br3 = new BufferedReader(r3);  
    System.out.println(br1.readLine());  
    System.out.println(br2.readLine());  
    System.out.println(br3.readLine());  
    br1.close();  
    br2.close();  
    br3.close();  
}
```

`BufferedReader` dekoriert einen `Reader`. Als zusätzliche Funktionalität bietet er die Möglichkeit an, eine komplette Textzeile zu lesen.

Auch die folgende Klasse `NumberReader` ist ein `Reader`, der einen anderen `Reader` nutzt:



```
package util;  
// ...  
import java.io.Reader;  
  
public class NumberReader extends Reader {  
    private final Reader reader;  
  
    private int currentChar = ' '  
  
    private final StringBuilder buf = new StringBuilder();  
  
    public NumberReader(Reader reader) {  
        this.reader = reader;  
    }  
    @Override  
    public void close() throws IOException {
```

```

        this.reader.close();
    }
    @Override
    public int read(char[] buf, int offset, int length)
        throws IOException {
        return this.reader.read(buf, offset, length);
    }

    public Number readNumber() throws IOException {
        this.skipWhitespace();
        if (this.currentChar == -1)
            return null;
        if (!Character.isDigit(this.currentChar))
            throw new NumberFormatException(
                "digit expected. But found: " + this.currentChar);
        this.buf.setLength(0);
        while(Character.isDigit(this.currentChar)) {
            this.buf.append((char) this.currentChar);
            this.next();
        }
        if(this.currentChar != '.') {
            return Integer.valueOf(this.buf.toString());
        }
        this.buf.append('.');
        this.next();
        while(Character.isDigit(this.currentChar)) {
            this.buf.append((char) this.currentChar);
            this.next();
        }
        return Double.valueOf(this.buf.toString());
    }

    private void skipWhitespace() throws IOException {
        while (Character.isWhitespace(this.currentChar)) {
            this.currentChar = this.reader.read();
        }
    }

    private void next() throws IOException {
        this.currentChar = this.reader.read();
    }
}

```

Dem Konstruktor wird der "eigentliche" `Reader` übergeben. Dieser wird genutzt, um in der `read`-Methode das nächste Zeichen zu lesen. Da `NumberReader` seinerseits von `Reader` abgeleitet ist und `Reader` noch abstrakt ist, muss genau eben diese `read`-Methode implementiert werden – und die `close`-Methode (auch diese delegiert an den "eigentlichen" `Reader`).

Eine Demo-Anwendung:

Die Datei `numbers.text` enthält folgende Zeile:

```
42  3.14      77  2.71
```

```
static void demo2() throws IOException {

    final Reader r1 = new InputStreamReader(
        new FileInputStream("src/numbers.txt"));

    final Reader r2 = new StringReader(" 123 456 789");

    final NumberReader nr1 = new NumberReader(r1);
    final NumberReader nr2 = new NumberReader(r2);

    Number number;
    number = nr1.readNumber();
    while (number != null) {
        System.out.println(number
            + " (" + number.getClass().getSimpleName() + ")");
        number = nr1.readNumber();
    }

    number = nr2.readNumber();
    while (number != null) {
        System.out.println(number
            + " (" + number.getClass().getSimpleName() + ")");
        number = nr2.readNumber();
    }

    nr1.close();
    nr2.close();
}
```

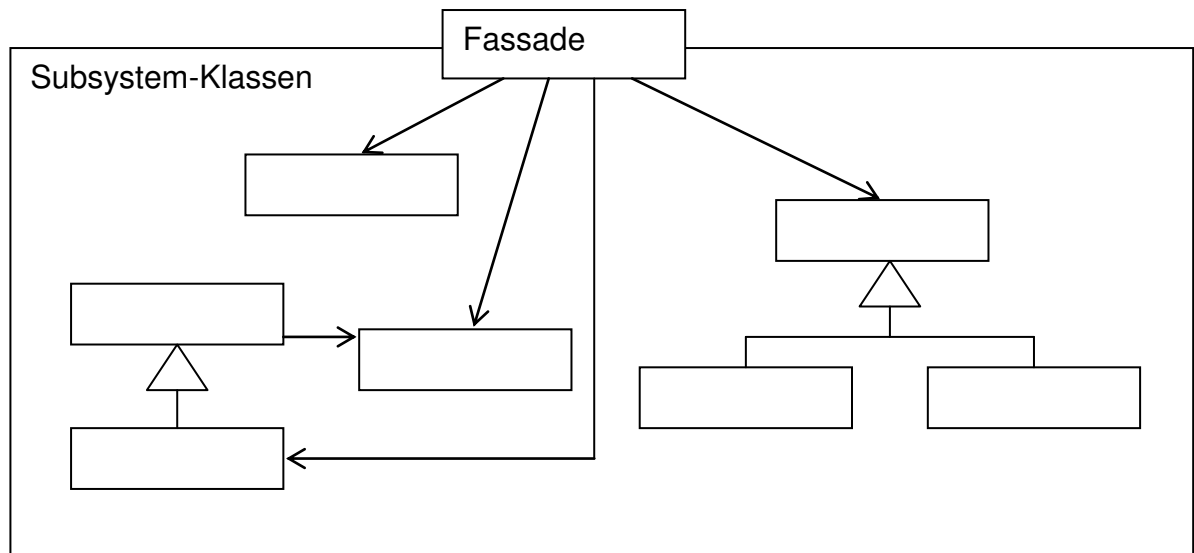
Die `close`-Methoden sollten natürlich in einem `finally`-Block aufgerufen werden (bzw. man sollte den `Resource-try` verwenden...)

Die Ausgaben:

```
42 (Integer)
3.14 (Double)
77 (Integer)
2.71 (Double)
123 (Integer)
456 (Integer)
789 (Integer)
```

3.4 Facade

"Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht."
(Gamma, 189)

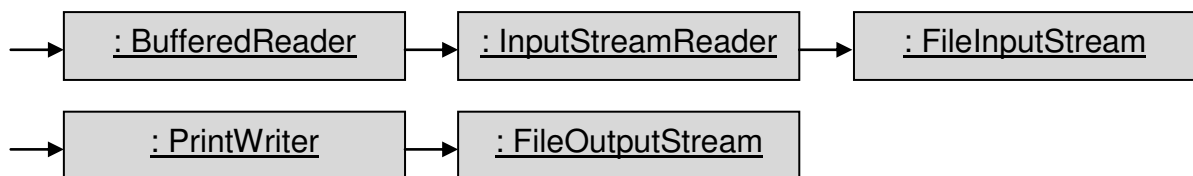


3.4.1 Problem

Eine Textdatei soll zeilenweise eingelesen werden. Jede Zeile soll in etwas anderer Form wieder ausgegeben werden: zunächst soll die Länge der Zeile und dann diese selbst (aber in Upper-Case) ausgegeben werden.

Man benötigt eine Reihe von Objekten:

- einen `FileInputStream`
- einen `InputStreamReader`
- einen `BufferedReader`
- einen `FileOutputStream`
- einen `PrintWriter`



Hier die Anwendung:

```
package appl;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;

public class Application {
    public static void main(String[] args) {

        final String inputFileName = "src/appl/Application.java";
        final String outputFileName = "upper.txt";

        final List<String> inputLines = new ArrayList<>();
        final List<String> outputLines = new ArrayList<>();

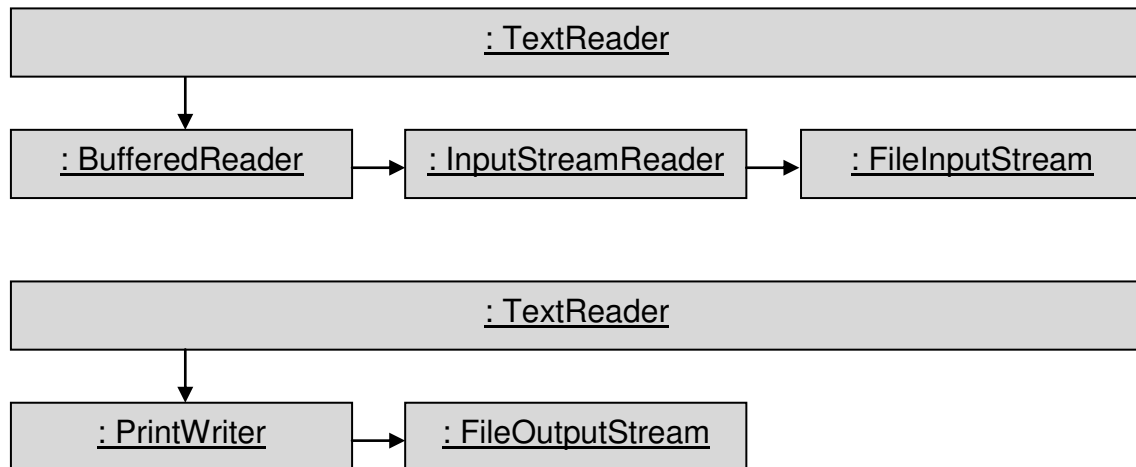
        try(final FileInputStream fis =
            new FileInputStream(inputFileName);
            final InputStreamReader isr =
                new InputStreamReader(fis);
            final BufferedReader br =
                new BufferedReader(isr)) {
            for (String line = br.readLine();
                line != null;
                line = br.readLine()) {
                inputLines.add(line);
            }
        }
    }
}
```

```
        }  
    }  
    catch(final Exception e) {  
        throw new RuntimeException(e);  
    }  
  
    inputLines.forEach(  
        line -> outputLines.add(line.toUpperCase()));  
  
    try(final FileOutputStream fos =  
        new FileOutputStream(outputFileName);  
        final PrintWriter pw =  
            new PrintWriter(fos)) {  
        outputLines.forEach(  
            line -> pw.println(line.length() + " : " + line));  
    }  
    catch(final Exception e) {  
        throw new RuntimeException(e);  
    }  
}
```

Was muss man nicht allein importieren...

3.4.2 Lösung

Sollten viele Probleme der Anwendung darin bestehen, eine Textdatei zeilenweise zu lesen und als Resultat wiederum eine Textdatei zu erzeugen, bieten sich die Klassen `TextReader` und `TextWriter` an, die als Facade fungieren.



Der `TextReader`:

```
package util;

import java.io.BufferedReader;
import java.io.Closeable;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Iterator;

public class TextReader implements Closeable, Iterable<String> {
    private final BufferedReader br;

    public TextReader(String filename) {
        try {
            this.br = new BufferedReader(
                new InputStreamReader(
                    new FileInputStream(filename)));
        }
        catch (final IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public void close() {
        try {
            this.br.close();
        }
        catch (final IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
```



```

public Iterator<String> iterator() {
    return new Iterator<String>() {
        String line = null;
        @Override
        public boolean hasNext() {
            try {
                this.line = TextReader.this.br.readLine();
            }
            catch (final IOException e) {
                throw new RuntimeException(e);
            }
            return this.line != null;
        }
        @Override
        public String next() {
            return this.line;
        }
    };
}

```

Der `TextReader` kapselt drei Dinge: den `FileInputStream`, den `InputStreamReader` und den `BufferedReader`. Er transformiert zudem alle `IOExceptions` in `RuntimeExceptions` (die bekanntlich abgefangen werden können, aber nicht an Ort und Stelle abgefangen werden müssen). Das macht es für eine Anwendung wesentlich einfacher, Zeile für Zeile einer Eingabedatei zu lesen.

Hinweis: (moderne Frameworks (z.B. Spring, JPA) transformieren häufig checked `Exceptions`, die ein Basissystem wirft, in `RuntimeExceptions`...)

Neben dem "externen" `Iterator`, den ein `TextReader` liefern kann, könnte auch noch ein "interner" `Iterator` definiert werden (eine `foreach`-Methode).

Der `TextWriter`:

```

package util;

import java.io.Closeable;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;

public class TextWriter implements Closeable {
    private final PrintWriter pw;
    public TextWriter(String filename) {
        try {
            this.pw = new PrintWriter(new FileOutputStream(filename));
        }
        catch (final IOException e) {
            throw new RuntimeException(e);
        }
    }
    public void writeln(String line) {
        this.pw.println(line);
    }
}

```

```
@Override
public void close() {
    this.pw.close();
}
}
```

Der `TextWriter` kapselt die Benutzung der Klassen `FileOutputStream` und `PrintWriter`. Auch er transformiert `IOExceptions` in `RuntimeExceptions`.

Die Applikation wird dann schlanker:

```
package appl;

import java.util.ArrayList;
import java.util.List;
import util.TextReader;
import util.TextWriter;

public class Application {
    public static void main(String[] args) {

        final String inputFileName = "src/appl/Application.java";
        final String outputFileName = "upper.txt";

        final List<String> inputLines = new ArrayList<>();
        final List<String> outputLines = new ArrayList<>();

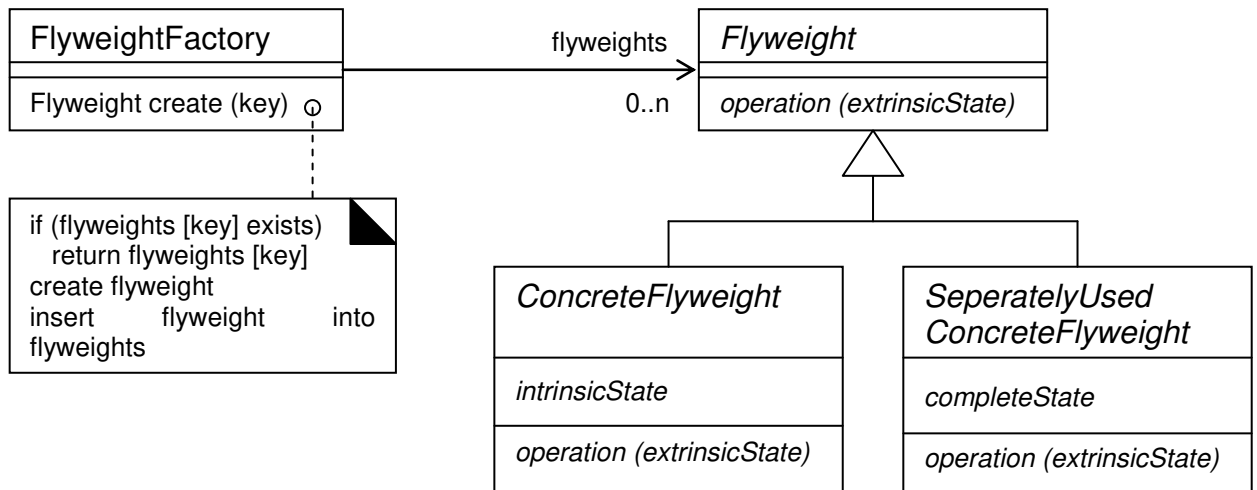
        try(final TextReader reader = new TextReader(inputFileName)) {
            for (final String line : reader)
                inputLines.add(line);
        }

        inputLines.forEach(
            line -> outputLines.add(line.toUpperCase());
        );

        try(final TextWriter writer = new TextWriter(outputFileName))
        {
            outputLines.forEach(
                line -> writer.writeln(line.length() + " : " + line));
        }
    }
}
```

3.5 Flyweight

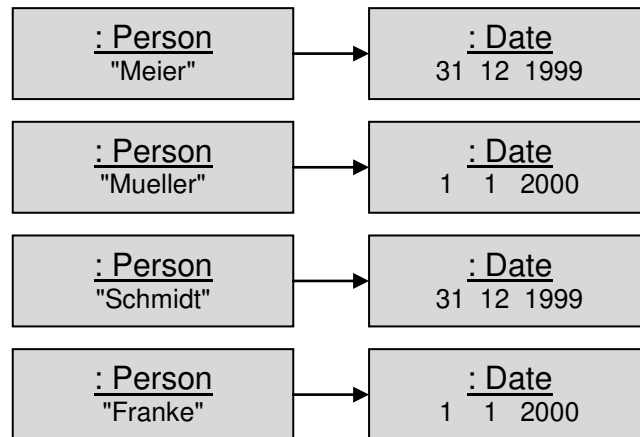
"Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient verwalten zu können." (Gamma, 198)



3.5.1 Problem

In einem Standesamt werden Personen registriert. Jede Person hat ein Geburtsdatum.

Datums (ja: das Wort gibt's wirklich) werden durch `Date`-Objekte repräsentiert:



```
package appl;

public class Date {

    private final int day;
    private final int month;
    private final int year;

    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public int getDay()    { return this.day;  }
    public int getMonth()  { return this.month; }
    public int getYear()   { return this.year; }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + this.day;
        result = prime * result + this.month;
        result = prime * result + this.year;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
```

```

        if (this.getClass() != obj.getClass())
            return false;
        final Date other = (Date) obj;
        return this.day == other.day &&
            this.month == other.month &&
            this.year == other.year;
    }

    @Override
    public String toString() {
        return "Date [day=" + this.day + ", month=" +
            this.month + ", year=" + this.year + "]";
    }
}

```

Man beachte, dass `Date`-Objekt konstant sind.

Personen werden durch `Person`-Objekte repräsentiert:

```

package appl;

public class Person {

    private final String name;
    private final Date birthdate;

    public Person(String name, Date birthdate) {
        this.name = name;
        this.birthdate = birthdate;
    }

    public String getName()      { return this.name; }
    public Date getBirthdate()  { return this.birthdate; }

    @Override
    public String toString() {
        return "Person [name=" + this.name +
            ", birthdate=" + this.birthdate + "]";
    }
}

```

Und hier eine kleine Anwendung:

```

package appl;
// ...
public class Application {

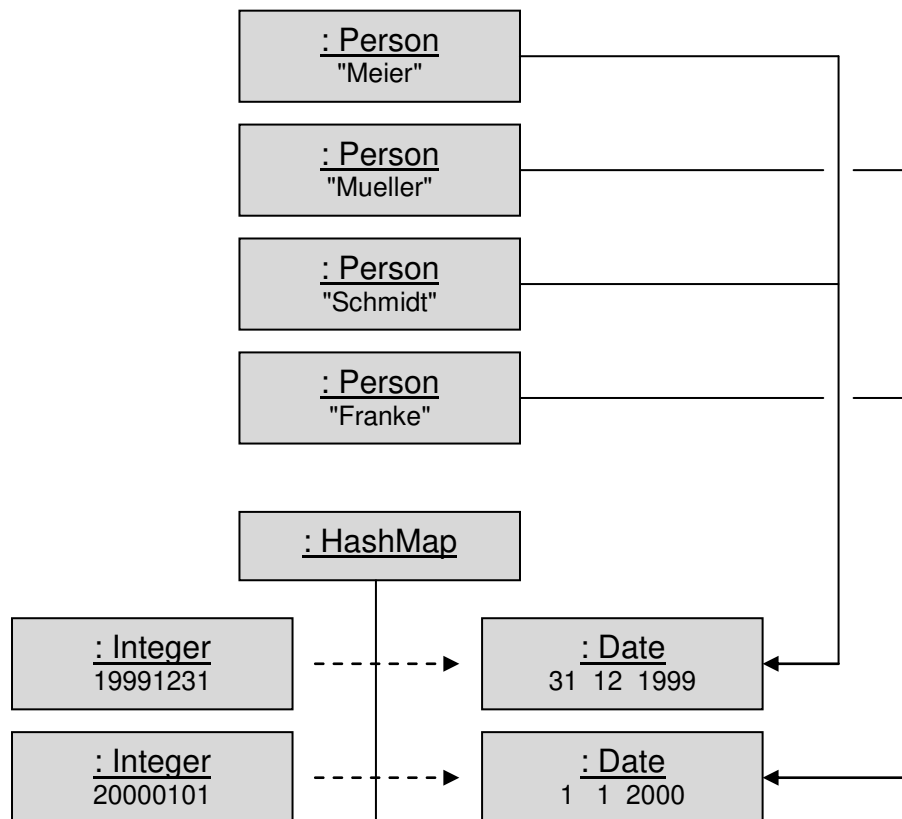
    public static void main(String[] args) {
        final List<Person> list = new ArrayList<Person>();
        list.add(new Person("Meier", new Date(31,12, 1999)));
        list.add(new Person("Mueller", new Date(1,1, 2000)));
        list.add(new Person("Schmidt", new Date(31,12, 1999)));
        list.add(new Person("Franke", new Date(1,1, 2000)));
        for(final Person p : list)
            System.out.println(p);
    }
}

```

Es gibt zwei Personen, die am 31.12.1999 geboren sind; und zwei weitere Personen, die am 1.1.2000 geboren sind. Insgesamt gibt's vier `Date`-Objekte – obwohl eigentlich zwei `Date`-Objekt ausreichen würden: Personen, die dasselbe Geburtsdatum haben, können sich auch auf ein und dasselbe `Date`-Objekt beziehen (dann muss allerdings vorausgesetzt werden, dass `Date`-Objekte konstant sind – aber genau das ist ja der Fall).

Es wäre also schön, wenn man sicherstellen könnte, dass zu einem Datum auch maximal ein einziges `Date`-Objekt existiert.

3.5.2 Lösung



Wir erweitern die Klasse `Date` wie folgt:

```

package appl;
// ...
public class Date {

    private static Map<Integer, Date> cache = new HashMap<>();

    public static int size() {
        return cache.size();
    }

    public static Date get(int day, int month, int year) {
        final int key = day + 100 * month + 10000 * year;
        synchronized(cache) {
            Date value = cache.get(key);
            if (value == null) {
                value = new Date(day, month, year);
                cache.put(key, value);
            }
            return value;
        }
    }

    private final int day;
  
```

```
private final int month;
private final int year;

private Date(int day, int month, int year) { ... }

public int getDay() { ... }
public int getMonth() { ... }
public int getYear() { ... }

@Override
public String toString() {
    return "Date [day=" + this.day + ", month=" +
        this.month + ", year=" + this.year + "]";
}
}
```

Der Konstruktor von `Date` ist privatisiert worden – somit kann außerhalb der `Date`-Klasse kein `Date`-Objekt mehr erzeugt werden.

Die Klasse enthält zudem ein privates statisches (also: globales) `HashMap`-Objekt, in welchem `Integer` auf `Date`-Objekte abgebildet werden. Diese `HashMap` fungiert als Cache, in welchem alle existierenden `Date`-Objekte gespeichert werden.

Weiterhin existiert eine öffentliche statische Methode `get`. Diese berechnet aufgrund der ihr übergebenen Parameter einen eindeutigen `int`-Schlüssel und schaut nach, ob die `Map` zu diesem Schlüssel bereits ein `Date`-Objekt enthält. Wenn ja, dann wird das im Cache bereits enthaltene Objekt zurückgeliefert. Ansonsten wird das neue `Date`-Objekt unter dem berechneten Schlüssel in den Cache eingefügt und zurückgeliefert.

Man beachte, dass die `get`-Methode synchronisiert ist (`synchronized`). Man beachte weiterhin, dass die Methoden `equals` und `hashCode` nun nicht mehr überschrieben überschrieben werden müssen(!)

Die `get`-Methode könnte man mit den Mitteln von Java-8 auch ein wenig kompakter schreiben:

```
public static Date get(int day, int month, int year) {
    final int key = day + 100 * month + 10000 * year;
    synchronized(cache) {
        return cache.computeIfAbsent(
            key, k -> new Date(day, month, year));
    }
}
```

(Man beachte, dass hier die `Map`-Methode `computeIfAbsent` benutzt wird – und nicht `putIfAbsent`!) Bei der Benutzung von `computeIfAbsent` könnte auch die `ConcurrentHashMap`-Klasse verwendet werden – wobei `synchronized` dann überflüssig würde.

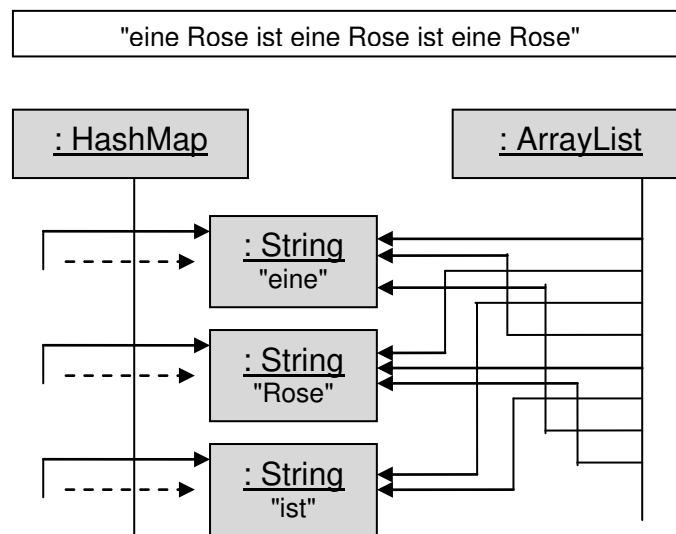
Hier eine Anwendung:

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {
        final List<Person> list = new ArrayList<Person>();
        list.add(new Person("Meier", Date.get(31,12, 1999)));
        list.add(new Person("Mueller", Date.get(1,1, 2000)));
        list.add(new Person("Schmidt", Date.get(31,12, 1999)));
        list.add(new Person("Franke", Date.get(1,1, 2000)));
        for(final Person p : list)
            System.out.println(p);
    }
}
```

Am Ende der `Main`-Methode wird der `Date`-Cache genau zwei Einträge besitzen – und jeweils zwei Personen teilen sich dasselbe `Date`-Objekt.

3.5.3 Ein String-Cache



In der folgenden Anwendung benutzen wir einen Cache für Strings:

```

package util;
// ...
public class StringCache {
    private final Map<String, String> cache = new HashMap<>();

    public String get(String obj) {
        synchronized(this.cache) {
            final String key = obj;
            String value = this.cache.get(key);
            if (value == null) {
                value = key;
                System.out.println("--> caching " + value);
                this.cache.put(key, value);
            }
            return value;
        }
    }

    public int size() {
        return this.cache.size();
    }
}
  
```

Die Anwendung:

```
package appl;
// ...
import util.StringCache;

public class Application {
    public static void main(String[] args) {
        final StringCache cache = new StringCache();
        final List<String> wordList = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(
            new InputStreamReader(
                new FileInputStream("src/input.txt"))) {
            final String input = reader.readLine();
            final String[] words = input.split(" ");
            for (final String word : words)
                wordList.add(cache.get(word));
        }
        catch (final Exception e) {
            throw new RuntimeException(e);
        }
        for (final String word : wordList)
            System.out.print(word + " ");
        System.out.println();
        System.out.println("Anzahl der Strings im cache = " + cache.size());
    }
}
```

Der Inhalt von input.txt:

eine rose ist eine rose ist eine rose

Die Ausgaben:

```
--> caching eine
--> caching rose
--> caching ist
eine rose ist eine rose ist eine rose
Anzahl der Strings im cache = 3
```

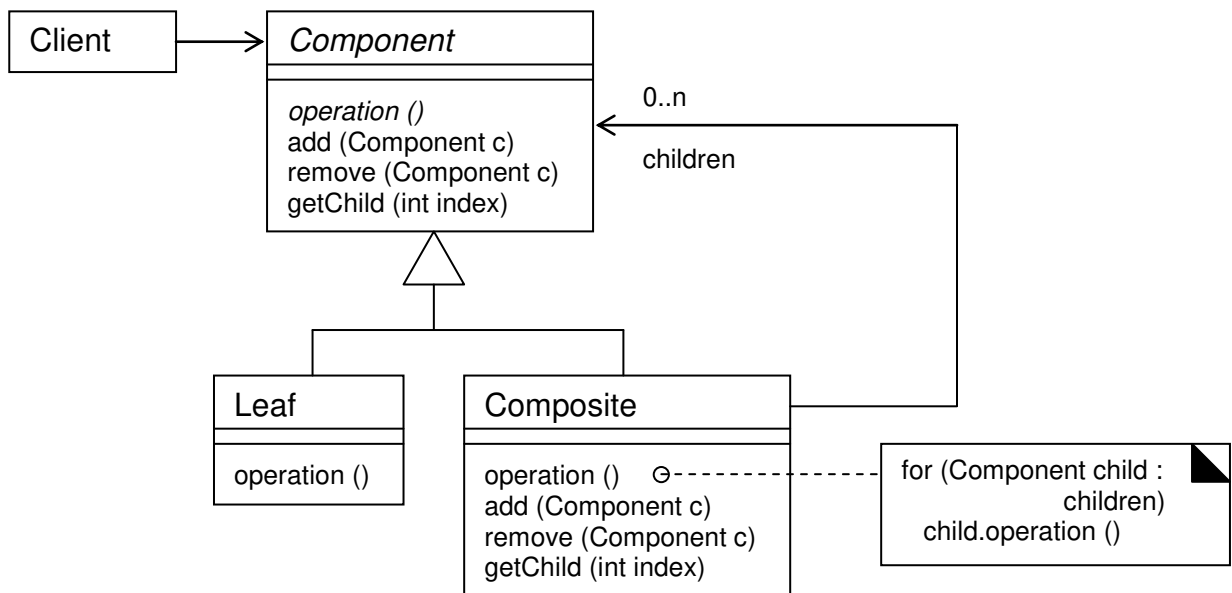
Aufgaben

Was hat es mit der String-Methode `intern()` auf sich?

Untersuchen Sie das Caching-Verhalten bei Integer-Objekten (untersuchen Sie die Methode `Integer.valueOf(...)`)

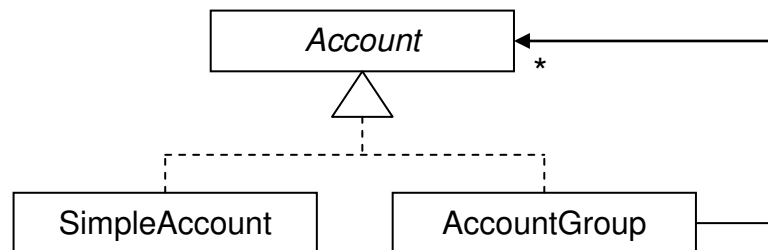
3.6 Composite

"Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln." (Gamma, 213)

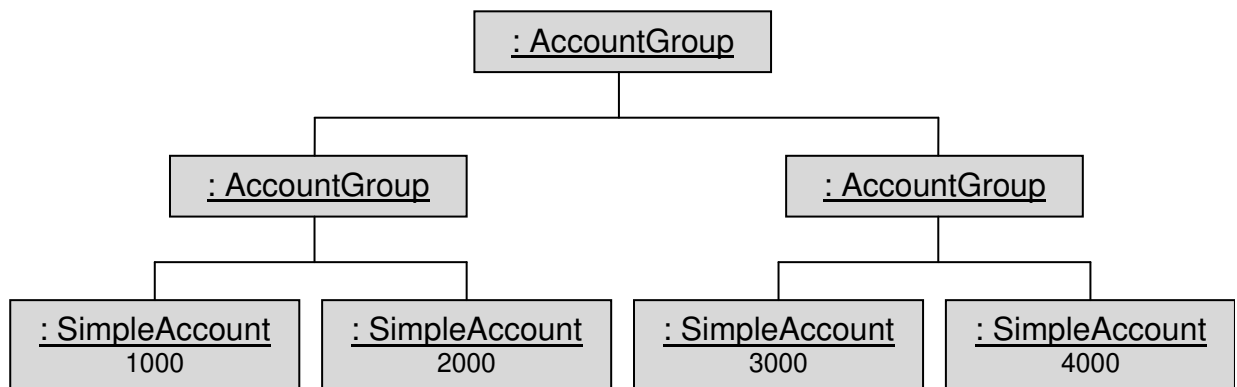


3.6.1 Problem

Ein Klassendiagramm:



Ein mögliches Objektdiagramm:



Ein Konto ist entweder ein elementares Konto oder ein Kontogruppe. Eine Kontogruppe kann ihrerseits wieder Konten bzw. weitere Kontogruppen enthalten. Konten können also einen Konto-Baum bilden.

Um elementare Konten und Kontogruppen vergleichbar zu machen, bietet sich zunächst die Definition des folgenden Interfaces an:

```

package appl;

public interface Account {
    public abstract double getBalance();
}
  
```

Jedes Konto hat einen Bestand. Der Bestand elementarer Konten ist in diesen selbst als Feld gespeichert; der Bestand von Kontogruppen ergibt sich aus der Summe der untergeordneten Konten/Kontogruppen – kann also rekursiv berechnet werden.

Von `Account` ist zunächst die Klasse elementarer Konten abgeleitet:

```
package appl;

public class SimpleAccount implements Account {

    private double balance;

    @Override
    public double getBalance() {
        return this.balance;
    }

    public void deposit(double value) {
        this.balance += value;
    }

    public void withdraw(double value) {
        this.balance -= value;
    }
}
```

Abgesehen davon, dass der Bestand eines `SimpleAccount`-Objekts abgefragt werden kann, können auf `SimpleAccount`-Objekte Ein- und Auszahlungen vorgenommen werden.

Die Klasse, deren Objekte Kontogruppen repräsentieren, ist wie folgt definiert:

```
package appl;
// ...
public class AccountGroup implements Account {

    private final List<Account> children = new ArrayList<>();

    @Override
    public double getBalance() {
        double sum = 0;
        for (final Account account : this.children)
            sum += account.getBalance();
        return sum;
    }

    public void add(Account account) {
        this.children.add(account);
    }

    public int getChildCount() {
        return this.children.size();
    }

    public Account getChild(int index) {
        return this.children.get(index);
    }
}
```

Auch `AccountGroup` implementiert `Account` (man kann also den Bestand einer `AccountGroup` abfragen). Zusätzlich können zu einer `AccountGroup` per `add` Objekte hinzugefügt werden, deren Klassen `Account`-kompatibel sind – also weitere `SimpleAccounts` oder `AccountGroups`. Die Methode `getChildCount` liefert die Anzahl der unmittelbaren Kinder (`SimpleAccounts` / `AccountGroups`) und `getChild` liefert das indexte Kind (als `Account`).

Eine kleine Anwendung:

```
package appl;

public class Application {

    public static void main(String[] args) {
        final AccountGroup g = new AccountGroup();
        final AccountGroup g1 = new AccountGroup();
        final AccountGroup g2 = new AccountGroup();
        final SimpleAccount a11 = new SimpleAccount();
        final SimpleAccount a12 = new SimpleAccount();
        final SimpleAccount a21 = new SimpleAccount();
        final SimpleAccount a22 = new SimpleAccount();
        a11.deposit(1000);
        a12.deposit(2000);
        a21.deposit(3000);
        a22.deposit(4000);
        g.add(g1);
        g.add(g2);
        g1.add(a11);
        g1.add(a12);
        g2.add(a21);
        g2.add(a22);

        System.out.println("\ng.Balance");
        System.out.println(g.getBalance());

        System.out.println("\nPrint(a11)");
        print(a11, "");

        System.out.println("\nPrint(g)");
        print(g, "");
    }

    static void print(Account account, String indent) {
        System.out.println(indent + account.getBalance());
        if (account instanceof AccountGroup) {
            final AccountGroup group = (AccountGroup)account;
            for (int i = 0; i < group.getChildCount(); i++) {
                final Account child = group.getChild(i);
                print(child, indent + "\t");
            }
        }
    }
}
```

Die Ausgabe des obigen Programms:

```
g.Balance
10000.0

Print(all)
1000.0

Print(g)
10000.0
    3000.0
        1000.0
        2000.0
    7000.0
        3000.0
        4000.0
```

Die `print`-Methode ist leider etwas problematisch. Sie ist rekursiv implementiert und muss daher für jedes Kind einer `AccountGroup` seinerseits wiederum die `print`-Methode aufrufen. Dazu muss zunächst einmal die Anzahl der Kinder ermittelt werden. Dies ist aber nur dann möglich, wenn `Account` – unter der Bedingung, dass es sich um eine `AccountGroup` handelt – eben auf `AccountGroup` gecastet wird – denn das Interface `Account` enthält keine `getChildCount`-Methode. Ebenso wenig enthält `Account` eine `getChild`-Methode, mittels derer sich das `index`-te Kind ermitteln lässt.

Es wäre schöner, wenn einerseits die `instanceof`-Abfrage und zweitens der `Downcast` nicht erforderlich wären – wenn man also `SimpleAccount`-Objekte genauso wie `AccountGroup`-Objekte behandeln könnte.

3.6.2 Lösung

Man erweitert das Interface `Account` um zwei weitere Methoden, deren Spezifikation auf dieser Ebene zunächst als "unsinnig" erscheint – nämlich um `getChildCount` und `getChild` (Methoden, die sich im nachhinein aber als sehr wertvoll herausstellen werden):

```
package appl;

public interface Account {
    public abstract double getBalance();
    public abstract int getChildCount();
    public abstract Account getChild(int index);
}
```

Die Klasse `SimpleAccount` muss dann natürlich zusätzlich diese beiden Methoden implementieren:

```
package appl;

public class SimpleAccount implements Account {

    private double balance;

    @Override
    public double getBalance() {
        return this.balance;
    }

    @Override
    public int getChildCount() {
        return 0;
    }

    @Override
    public Account getChild(int index) {
        throw new RuntimeException();
    }

    public void deposit(double value) {
        this.balance += value;
    }

    public void withdraw(double value) {
        this.balance -= value;
    }
}
```

`getChildCount` liefert 0; `getChild` wirft eine `RuntimeException`.

Und `AccountGroup` implementiert diese Methoden ohnehin.

Die Anwendung wird nun um einiges einfacher:

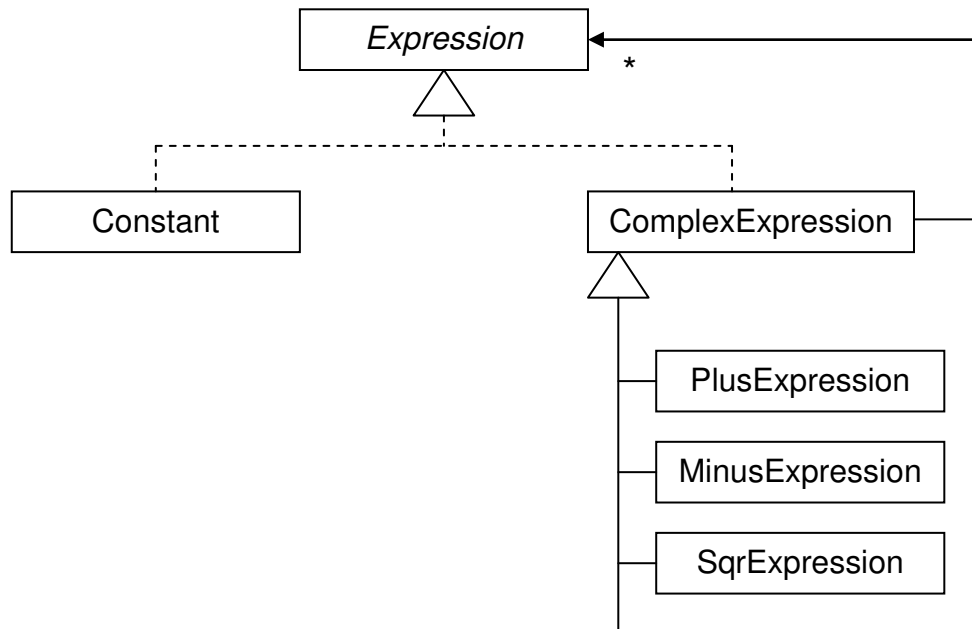
```
package appl;

public class Application {
    public static void main(String[] args) {
        // ...
    }

    static void print(Account account, String indent) {
        System.out.println(indent + account.getBalance());
        for (int i = 0; i < account.getChildCount(); i++) {
            final Account child = account.getChild(i);
            print(child, indent + "\t");
        }
    }
}
```

Die Anwendung benötigt nun einerseits keine dynamische Typabfrage (`instanceof`) mehr und zweitens keinen Downcast. `SimpleAccount`-Objekt können genauso behandelt werden wie `AccountGroup`-Objekte.

3.6.3 Expressions



Ausdrücke (Expressions) sind Konstrukte, welche einen Wert haben resp. deren Wert man berechnen kann. Ein Ausdruck kann ein einfacher Wert sein (eine Konstante) oder aber aus untergeordneten Ausdrücken und einer Berechnungsvorschrift bestehen.

Man betrachte z.B. den folgenden Ausdruck:

`22 + 33`

Ein solcher Ausdruck kann auch in folgender Form – als "Function-Call" – notiert werden:

`plus(22, 33)`

Es handelt sich um einen Ausdruck, der aus zwei untergeordneten Ausdrücken besteht, welche ihrerseits einfache Konstanten darstellen. Der folgende Ausdruck ist komplizierter:

`(22 + 33) * 2`

Auch dieser Ausdruck besteht aus zwei Teilausdrücken. Der erste Teilausdruck ist aber wiederum ein (komplexer) Ausdruck, welcher aus zwei Teilausdrücken besteht.

Derselbe Ausdruck als Function-Call:

`times (sum (22, 33), 2)`

Dieser Begriff einer Expression kann in folgendem Interface spezifiziert werden (wir beschränken uns hier auf Expressions, deren Wert vom Typ `double` ist):

```
package appl;

public interface Expression {
    public abstract double evaluate();
    public abstract int getChildCount();
    public abstract Expression getChild(int index);
}
```

`getChildCount` liefert die Anzahl der Sub-Expressions (oder 0: bei konstanten Ausdrücken); `getChild` liefert (sofern es sich um komplexe Expressions handelt) die `index`-te Sub-Expression. `evaluate` liefert den Wert der Expression.

Dieses Interface kann dann zunächst in einer Klasse implementiert werden, deren Objekte einfache Konstanten sind:

```
package appl;

public class Constant implements Expression {

    private final double value;

    public Constant(double value) {
        this.value = value;
    }

    @Override
    public double evaluate() {
        return this.value;
    }

    @Override
    public int getChildCount() {
        return 0;
    }

    @Override
    public Expression getChild(int index) {
        throw new RuntimeException();
    }

    @Override
    public String toString() {
        return String.valueOf(this.value);
    }
}
```

Man beachte, dass `Constant`-Objekte konstant sind. Man beachte weiterhin, dass `getChild` eine `RuntimeException` wirft.

Das Interface `Expression` kann dann in einer weiteren abstrakten Basisklasse implementiert werden:

```
package appl;
// ...
public abstract class ComplexExpression implements Expression {

    private final String name;
    private final Expression[] children;

    public ComplexExpression(String name, Expression... children) {
        this.name = name;
        this.children = children;
    }

    @Override
    public int getChildCount() {
        return this.children.length;
    }

    @Override
    public Expression getChild(int index) {
        return this.children [index];
    }

    @Override
    public String toString() {
        return this.name + Arrays.toString(this.children);
    }
}
```

Eine `ComplexExpression` repräsentiert eine komplexe Expression. Sie hat einen Namen ("plus", "minus" etc.) und untergeordnete Expressions (Konstanten oder wiederum komplexe Expressions). `getChildCount` liefert die Anzahl der untergeordneten Expressions zurück, `getChild` das index-te Argument.

Die einzige Methode, welche auf der Ebene dieser abstrakten Basisklasse noch nicht implementiert werden kann, ist `evaluate`.

Hier einige Ableitungen von `ComplexExpression`:

```
package appl;

public class PlusExpression extends ComplexExpression {

    public Plus(Expression e0, Expression e1) {
        super("plus", e0, e1);
    }

    @Override
    public double evaluate() {
        return this.getChild(0).evaluate() +
this.getChild(1).evaluate();
    }
}
```

```
package appl;

public class MinusExpression extends ComplexExpression {

    public MinusExpression(Expression e0, Expression e1) {
        super("minus", e0, e1);
    }

    @Override
    public double evaluate() {
        return this.getChild(0).evaluate() -
this.getChild(1).evaluate();
    }
}
```

```
package appl;

public class SqrExpression extends ComplexExpression {

    public SqrExpression(Expression e) {
        super("sqr", e);
    }

    @Override
    public double evaluate() {
        final double value = this.getChild(0).evaluate();
        return value * value;
    }
}
```

Und hier eine kleine Test-Anwendung:

```
package appl;

public class Application {
    public static void main(String[] args) {
        final Expression sqr = new SqrExpression(
            new Constant(2));
        final Expression minus = new MinusExpression(
            new Constant(5), new Constant(2));
        final Expression plus = new PlusExpression(
            minus, sqr);
        System.out.println(plus.evaluate());
        print(plus, "");
    }

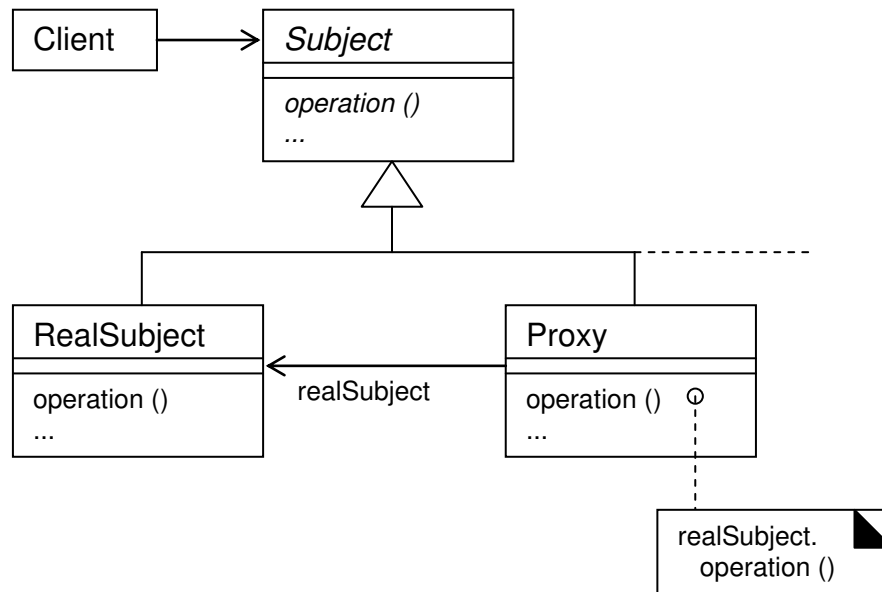
    static void print(Expression expression, String indent) {
        System.out.println(indent + expression);
        for (int i = 0; i < expression.getChildCount(); i++) {
            final Expression child = expression.getChild(i);
            print(child, indent + "\t");
        }
    }
}
```

Die Ausgaben des obigen Programms:

```
7.0
plus[minus[5.0, 2.0], sqr[2.0]]
    minus[5.0, 2.0]
        5.0
        2.0
    sqr[2.0]
        2.0
```

3.7 Proxy

"Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreter-objekts." (Gamma, 227)



Hinweis: sich eigentlich auf die Basisklasse `Subject` beziehen – und nicht auf `RealSubject`. Aber Gamma hat's halt so gezeichnet...

3.7.1 Problem

Ein Mathematik-Service kann die Summe und die Differenz zweier Zahlen berechnen. Er sollte seine Aktivitäten auch tracen können:

```
package appl;

public class MathService {

    public boolean mustTrace = false;

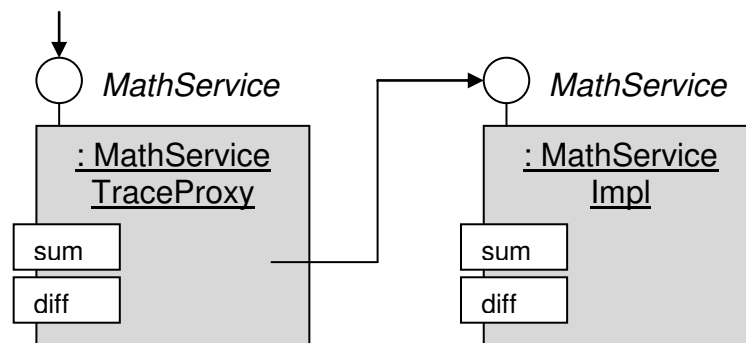
    public int sum(int x, int y) {
        if (this.mustTrace)
            System.out.println(">> sum(" + x + ", " + y + ")");
        final int result = x + y;
        if (this.mustTrace)
            System.out.println("<< sum(" + x + ", " + y + ") --> " + result);
        return result;
    }

    public int diff(int x, int y) {
        if (this.mustTrace)
            System.out.println(">> diff(" + x + ", " + y + ")");
        final int result = x - y;
        if (this.mustTrace)
            System.out.println("<< diff(" + x + ", " + y + ") --> " + result);
        return result;
    }
}
```

Eine "realistische" Klasse könnte neben dem hier dargestellten Trace-Aspekt um weitere Aspekte erweitert werden müssen: Notifications, Berechtigungen, Transaktions-Handling. Es handelt sich hier stets um Dinge, die mit der eigentlichen Natur der Sache (hier: Summen- und Differenzbildung) nichts zu tun haben. Implementiert man solche Aspekte zusammen mit der eigentlichen Funktionalität in derselben Klasse, wird diese leicht unübersichtlich – man sieht den Wald vor lauter Bäumen nicht mehr.

Es wäre wünschenswert, solche Aspekte aus der eigentlichen Klasse auszulagern.

3.7.2 Lösung



Wir beginnen zunächst mit einem Interface, welches die Funktionalität der eigentlichen Klasse spezifiziert:

```
package appl;

public interface MathService {
    public abstract int sum(int x, int y);
    public abstract int diff(int x, int y);
}
```

Die Klasse, welche die eigentliche Funktionalität enthält, implementiert dann dieses Interface – und beschränkt sich in ihrer Implementierung auf den Kern der Sache:

```
package appl;

public class MathServiceImpl implements MathService {

    @Override
    public int sum(int x, int y) {
        return x + y;
    }

    @Override
    public int diff(int x, int y) {
        return x - y;
    }
}
```

Das Interface kann dann in einer zweiten Klasse – in einer Proxy-Klasse – implementiert werden:

```
package appl;

public class MathServiceTraceProxy implements MathService {

    private final MathService target;

    public MathServiceTraceProxy(MathService target) {
        this.target = target;
    }

    @Override
```

```

public int sum(int x, int y) {
    System.out.println(">> sum(" + x + ", " + y + ")");
    final int result = this.target.sum(x, y);
    System.out.println("<< sum(" + x + ", " + y + ") --> " + result);
    return result;
}

@Override
public int diff(int x, int y) {
    System.out.println(">> diff(" + x + ", " + y + ")");
    final int result = this.target.diff(x, y);
    System.out.println("<< diff(" + x + ", " + y + ") --> " + result);
    return result;
}
}

```

Ein Proxy besitzt eine Referenz auf ein Objekt, dessen Klasse dasselbe Interface implementiert wie die Proxy-Klasse selbst (`target`). Diese Referenz wird sinnvollerweise mittels des Konstruktors initialisiert.

Die Methoden der Proxy-Klasse sind allesamt nach demselben Schema aufgebaut:

- Pre-Invoke (hier: Trace)
- Invoke (Aufruf der entsprechenden Methode auf das `target`-Objekt)
- Post-Invoke (hier: Trace)

Hier eine keine Anwendung:

```

package appl;

public class Application {

    public static void main(String[] args) {

        final MathService mathService =
            new MathServiceTraceProxy(
                new MathServiceImpl());

        final int sum = mathService.sum(40, 2);
        System.out.println(sum);
        final int diff = mathService.sum(80, 3);
        System.out.println(diff);
    }
}

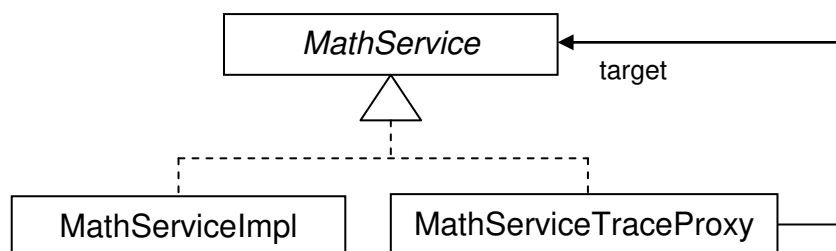
```

Nachdem der `MathServiceImpl` und sein Trace-Proxy erzeugt und miteinander verbunden wurden, kann die Anwendung den Trace-Proxy einfach als `MathService` behandeln – also genauso behandeln, wie auch der "richtige" Service behandelt werden könnte. Dem gesamte Rest der Anwendung kann es völlig gleichgültig sein, ob mit einem Proxy oder direkt mit dem "richtigen" Service gearbeitet wird.

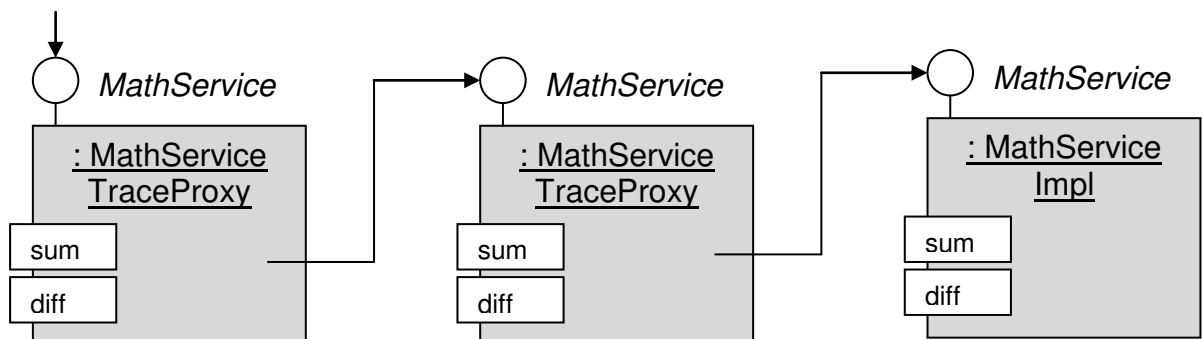
Die obige Anwendung erzeugt folgende Ausgaben:

```
>> sum(40, 2)
<< sum(40, 2) --> 42
42
>> sum(80, 3)
<< sum(80, 3) --> 83
83
```

Das Klassendiagramm:



Natürlich können wir nun auch "doppelt" tracen (Warum ist `target` vom Typ `MathService` (und nicht vom Typ `MathServiceImpl`)? Betrachten Sie noch einmal das Klassendiagramm von Gamma!)



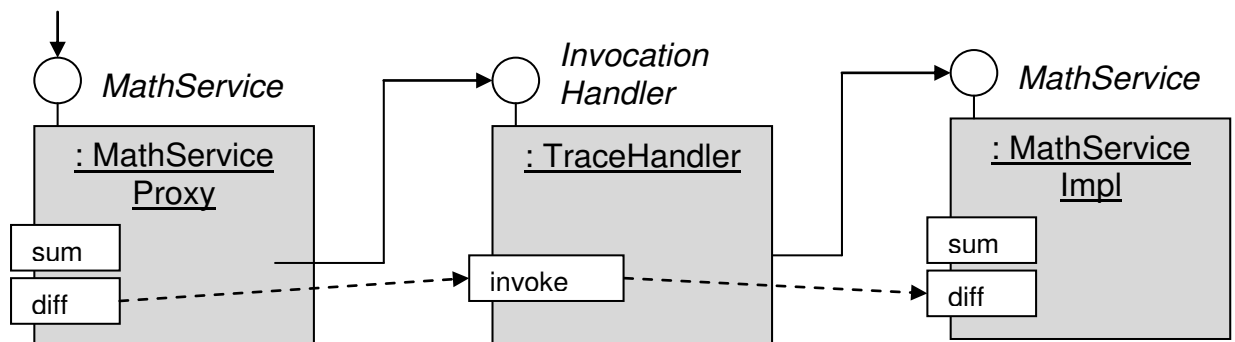
3.7.3 InvocationHandler

Angenommen, es gibt n fachliche Interfaces (wie z. B. `MathService`) – und m Aspekte (Tracen, Berechtigungen, Transaktionen, etc.). Wollte man nun für jedes fachliches Interface und jeden Aspekt eine Proxy-Klasse schreiben, müsste man $n \cdot m$ Proxy-Klassen schreiben. Es wäre besser, man müsste nur $m+n$ Klassen schreiben. (Und es wäre noch angenehmer, wenn man nur m Klassen schreiben müsste – für jeden Aspekt genau eine einzige Klasse...)

Die Lösung besteht darin, die `MathServiceTraceProxy`-Klasse in zwei Klassen aufzuspalten (der Name `MathServiceTraceProxy` ist ohnehin zu lang...): in die Klasse `MathServiceProxy` und einen `TraceHandler`. Die Klasse `TraceHandler` implementiert ein technisches Interface, und zwar ein Standard-Interface von Java:

```
package java.lang.reflect;

public interface InvocationHandler {
    Object invoke(Object proxy, Method method, Object[] parameters);
}
```



Die Klasse `TraceHandler`:

```
package util;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.Arrays;

public class TraceHandler implements InvocationHandler {

    private final Object target;

    public TraceHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(
        Object proxy, Method method, Object[] parameters)
        throws Throwable {

        System.out.println(">> " + method.getName() + " " +
```

```

        Arrays.toString(parameters));

    try {
        Object result;
        if (this.target instanceof InvocationHandler)
            result = ((InvocationHandler) this.target).invoke(
                proxy, method, parameters);
        else
            result = method.invoke(this.target, parameters);

        System.out.println("<< " + method.getName() + " " +
            Arrays.toString(parameters) + " --> " + result);
        return result;
    }
    catch (final InvocationTargetException e) {
        throw e.getTargetException();
    }
}
}

```

Dem `TraceHandler` wird das eigentliche Ziel-Objekt übergeben – in der Form einer `Object`-Referenz! Der `invoke`-Methode wird ein `Method`-Objekt und die Aufrufparameter (in Form eines `Object`-Arrays) übergeben. Die `invoke`-Methode kann dann zunächst den Pre-Invoke-Schritt implementieren (Trace des Einstiegs in die Methode). Dann kann mittels des `Method`-Objekts die von diesem `Method`-Objekt beschriebene Methode auf das `target`-Objekt aufgerufen werden – mittels der `invoke`-Methode der Klasse `Method`. Schließlich kann der Post-Invoke-Schritt implementiert werden (Trace des Ausstiegs aus der Methode).

Die Fallunterscheidung in der `invoke`-Methode ermöglicht es, mehrere `InvocationHandler` in Reihe zu schalten. Im Falle, dass hinter einem `InvocationHandler` ein weiterer `InvocationHandler` liegt, wird dessen `invoke`-Methode aufgerufen – ansonsten die "eigentliche" Methode des "eigentlichen" Objekts.

Der `MathServiceProxy` beschränkt sich nun darauf, die `invoke`-Methode irgendeines Handler-Objekts aufzurufen (eines Objekts, dessen Klasse das Interface `InvocationHandler` implementiert):

```

package appl;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class MathServiceProxy implements MathService {

    private final InvocationHandler handler;

    public MathServiceProxy(InvocationHandler handler) {
        this.handler = handler;
    }

    @Override

```

```
public int sum(int x, int y) {
    try {
        final Method method = MathService.class.getMethod(
            "sum", int.class, int.class);
        final Object result = this.handler.invoke(
            this, method, new Object[] { x, y });
        return (Integer)result;
    }
    catch(final Throwable e) {
        throw new RuntimeException(e);
    }
}

@Override
public int diff(int x, int y) {
    try {
        final Method method = MathService.class.getMethod(
            "diff", int.class, int.class);
        final Object result = this.handler.invoke(
            this, method, new Object[] { x, y });
        return (Integer)result;
    }
    catch(final Throwable e) {
        throw new RuntimeException(e);
    }
}
}
```

Hier eine Applikation:

```
package appl;

import util.TraceHandler;

public class Application {

    public static void main(String[] args) {

        final MathService mathService =
            new MathServiceProxy(
                new TraceHandler(
                    new MathServiceImpl()));

        final int sum = mathService.sum(40, 2);
        System.out.println(sum);
        final int diff = mathService.sum(80, 3);
        System.out.println(diff);
    }
}
```

Man erkennt: zwischen dem `MathServiceProxy` und dem "richtigen" `MathService`-Objekt liegt der `TraceHandler`. Die Klasse `MathServiceProxy` ist völlig unabhängig geworden vom aktuellen Aspekt – und die `TraceHandler`-Klasse ist völlig unabhängig vom fachlichen Interface. Für jeden Aspekt muss nun also genau eine Handler-Klasse geschrieben werden – und für jedes fachliche Interface genau eine Proxy-Klasse (n+m).

3.7.4 DynamicProxy

Die Proxy-Klasse, die im letzten Abschnitt implementiert wurde, kann automatisch aufgrund des zu implementierenden Interfaces generiert werden. Wir könnten einen Generator schreiben, der irgendein Interface per Reflection analysiert und aufgrund dieser Analyse dann die Proxy-Klasse generiert.

Oder wir benutzen einen Generator, den Java bereits von Haus aus mitbringt – den Dynamic-Proxy-Generator. Dieser wird zur Laufzeit gestartet und erzeugt unmittelbar den Bytecode der Proxy-Klasse. Der Generator ist in der `Proxy`-Klasse des `Reflection-Packages` implementiert:

```
package appl;

import java.lang.reflect.Proxy;
import util.TraceHandler;

public class Application {

    public static void main(String[] args) {

        final TraceHandler handler =
            new TraceHandler(new MathServiceImpl());

        final MathService mathService =
            (MathService) Proxy.newProxyInstance(
                ClassLoader.getSystemClassLoader(),
                new Class<?>[] { MathService.class },
                handler);

        final int sum = mathService.sum(40, 2);
        System.out.println(sum);
        final int diff = mathService.sum(80, 3);
        System.out.println(diff);
    }
}
```

An `newProxyInstance` wird zunächst ein `ClassLoader` übergeben. Dann wird das zu implementierende Interface übergeben (genauer: ein Array der zu implementierenden Interfaces). Schließlich wird ein `InvocationHandler` übergeben. Aufgrund des zu implementierenden Interfaces kann die Proxy-Klasse generiert werden; dann kann diese Klasse instanziiert werden, wobei der `InvocationHandler` übergeben wird. Die `newProxyInstance`-Methode liefert schließlich die Referenz auf das erzeugte Proxy-Objekt zurück. (Die generierten Klassen haben Namen wie `$Proxy0`, `$Proxy1` etc.)

Sofern die Proxy-Klasse nur ein einziges Interfaces implementieren soll und der System-ClassLoader genutzt werden soll, können wir folgende kleine Convenience-Methode nutzen:

```
package util;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

public class SimpleProxy {

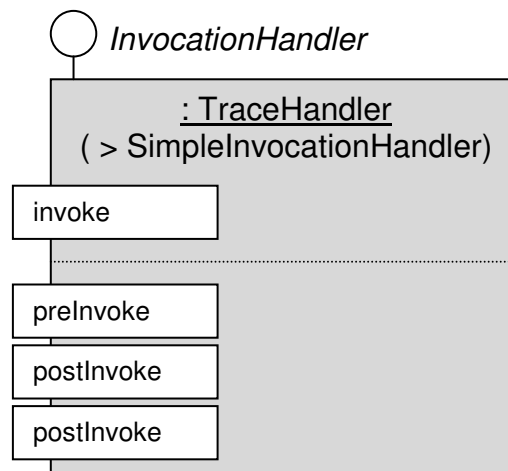
    public static <T> T create(Class<?> iface, InvocationHandler
handler) {
        return (T) Proxy.newProxyInstance(
            ClassLoader.getSystemClassLoader(),
            new Class<?>[] { iface },
            handler);
    }
}
```

Hier die Anwendung dieser Methode:

```
final TraceHandler handler =
    new TraceHandler(new MathServiceImpl());

final MathService mathService =
    SimpleProxy.create(MathService.class, handler);
```

3.7.5 SimpleInvocationHandler



Wir können das `InvocationHandler`-Interface in einer abstrakten Klasse implementieren:

```

package util;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public abstract class SimpleInvocationHandler
    implements InvocationHandler {

    private final Object target;

    public SimpleInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public final Object invoke(
        Object proxy, Method method, Object[] parameters)
        throws Throwable {

        this.preInvoke(method, parameters, this.target);
        try {
            Object result;
            if (this.target instanceof InvocationHandler)
                result = ((InvocationHandler) this.target).invoke(
                    proxy, method, parameters);
            else
                result = method.invoke(this.target, parameters);

            this.postInvoke(method, parameters, this.target, result);

            return result;
        }
        catch (final InvocationTargetException e) {
            this.postInvoke(method, parameters, this.target, e);
        }
    }
}
  
```

```
        throw e.getTargetException();
    }
    catch (final Throwable e) {
        this.postInvoke(method, parameters, this.target, e);
        throw e;
    }
}
protected void preInvoke(Method method, Object[] parameters,
    Object target) {
}
protected void postInvoke(Method method, Object[] parameters,
    Object target, Object result) {
}
protected void postInvoke(Method method, Object[] parameters,
    Object target, Throwable exception) {
}
}
```

Die `Invoke`-Methode implementiert auch hier den Dreisatz: Pre-Invoke, Invoke und Post-Invoke – aber derart, dass einfach an Methoden delegiert, die in dieser Basisklasse einfach leer implementiert sind: an die `preInvoke`- resp. an die beiden `postInvoke`-Methoden. Diese können in einer abgeleiteten Klasse überschrieben werden. (Dies ist eine Anwendung des Template-Method-Patterns.)

Von dieser allgemein verwendbaren Klasse kann dann für jeden Aspekt eine bestimmte Handler-Klasse abgeleitet werden. Z.B. die Klasse `TraceHandler`:

```
package util;

import java.lang.reflect.Method;
import java.util.Arrays;

public class TraceHandler extends SimpleInvocationHandler {
    public TraceHandler(Object target) {
        super(target);
    }
    @Override
    protected void preInvoke(Method method, Object[] parameters,
        Object target) {
        System.out.println(">> " + method.getName() + " " +
            Arrays.toString(parameters));
    }
    @Override
    protected void postInvoke(Method method, Object[] parameters,
        Object target, Object result) {
        System.out.println("<< " + method.getName() + " " +
            Arrays.toString(parameters) + " --> " + result);
    }
}
```


4

Verhaltensmuster

4.1	Command	4-6
4.1.1	Problem	4-7
4.1.2	Lösung	4-9
4.1.3	History	4-13
4.1.4	CommadFactory	4-18
4.2	Observer	4-20
4.2.1	Problem	4-21
4.2.2	Lösung	4-23
4.2.3	Lambdas	4-26
4.2.4	Vereinfachung	4-28
4.2.5	Observer/Observable	4-29
4.3	Visitor	4-32
4.3.1	Problem	4-33
4.3.2	Lösung	4-37
4.3.3	Dispatch mit Reflection	4-41
4.3.4	Dispatch mit Consumer	4-44
4.3.5	Lambdas	4-47
4.3.6	Swing-Beispiel	4-48
4.4	Interpreter	4-51
4.4.1	Problem	4-52
4.4.2	Lösung	4-54

4.4.3	Bau einer Datenstruktur	4-56
4.4.4	BeanFactory-Beispiel.....	4-59
4.5	Iterator	4-66
4.5.1	Problem	4-67
4.5.2	Lösung	4-70
4.5.3	Das Interface Iterable.....	4-74
4.5.4	Interner Iterator	4-76
4.5.5	Tree-Iterator	4-79
4.6	Memento	4-81
4.6.1	Problem	4-82
4.6.2	Lösung	4-85
4.6.3	Serialisierung	4-88
4.6.4	Ein grafischer Editor.....	4-90
4.7	Template Method.....	4-91
4.7.1	Problem	4-92
4.7.2	Lösung	4-94
4.7.3	Delegation via Interface	4-98
4.7.4	Erweiterung der Delegation	4-101
4.7.5	Ein Gruppenwechsel-Prozessor	4-103
4.8	Strategy	4-107
4.8.1	Problem	4-108
4.8.2	Lösung	4-110
4.8.3	Lambdas	4-112
4.8.4	XYLayout	4-113
4.8.5	Functions	4-115
4.8.6	Circuits.....	4-120
4.9	Mediator	4-126
4.9.1	Problem	4-127
4.9.2	Lösung	4-129
4.9.3	EventBus	4-131
4.9.4	Queue	4-137
4.10	State	4-142
4.10.1	Problem	4-143
4.10.2	Lösung	4-145
4.10.3	Stack.....	4-148
4.10.4	FigureEditor	4-151
4.10.5	Dispatcher.....	4-153
4.11	Chain of Responsibility	4-158
4.11.1	Problem	4-159
4.11.2	Lösung	4-160

4.11.3	JDBC-Driver und DriverManager	4-164
--------	-------------------------------------	-------

4 Verhaltensmuster

"Verhaltensmuster befassen sich mit Algorithmen und der Zuweisung von Zuständigkeiten zu Objekten. Verhaltensmuster beschreiben nicht nur Muster von Objekten und Klassen, sondern auch die Muster der Interaktion zwischen ihnen. Diese Muster beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachzuvollziehen sind. Sie lenken unsere Konzentration weg vom Kontrollfluss hin zu der Art und Weise, wie die Objekte miteinander interagieren.

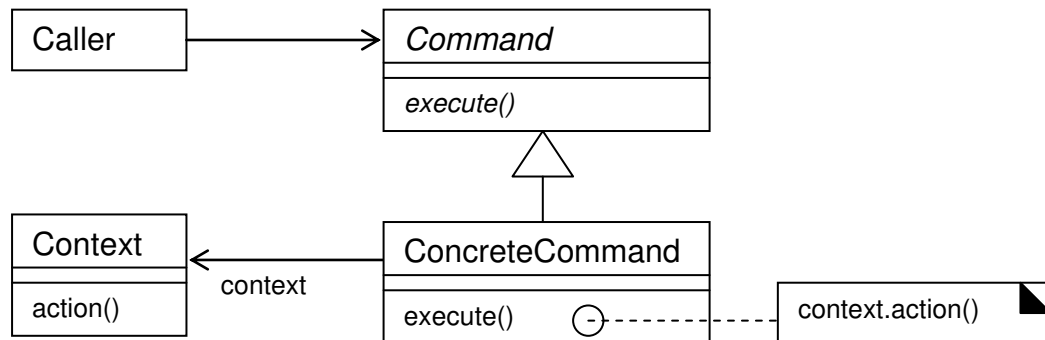
Klassenbasierte Verhaltensmuster verwenden Vererbung, um das Verhalten unter den Klassen zu verteilen...

Objektbasierte Verhaltensmuster verwenden Objektkomposition anstelle von Vererbung."

(Gamma, 243)

4.1 Command

"Kapsle einen Befehl als Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Queue zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen." (Gamma, 245)



4.1.1 Problem

Ein beispielhafter Dialog mit einem Taschenrechner (Ausgaben sind fett gedruckt):

```
add n | subtract n | get | end :
add 20
subtract 5
Bad command
subtract 5
get
15
end
```

Eine prozedurale Implementierung des Taschenrechners:

```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        System.out.println("add n | subtract n | get | end : ");
        int value = 0;
        String input;
        final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        while (! (input = reader.readLine()).equals("end")) {
            input = input.trim();
            if (input.length() == 0)
                continue;
            final String[] tokens = input.split(" ");
            value = executeCommand(tokens, value);
        }
    }

    static int executeCommand(String[] tokens, int value) {
        switch (tokens[0]) {
            case "add":
                return executeAddCommand(tokens, value);
            case "subtract":
                return executeSubtractCommand(tokens, value);
            case "get":
                return executeGetCommand(tokens, value);
            default:
                System.out.println("Bad command");
                return value;
        }
    }

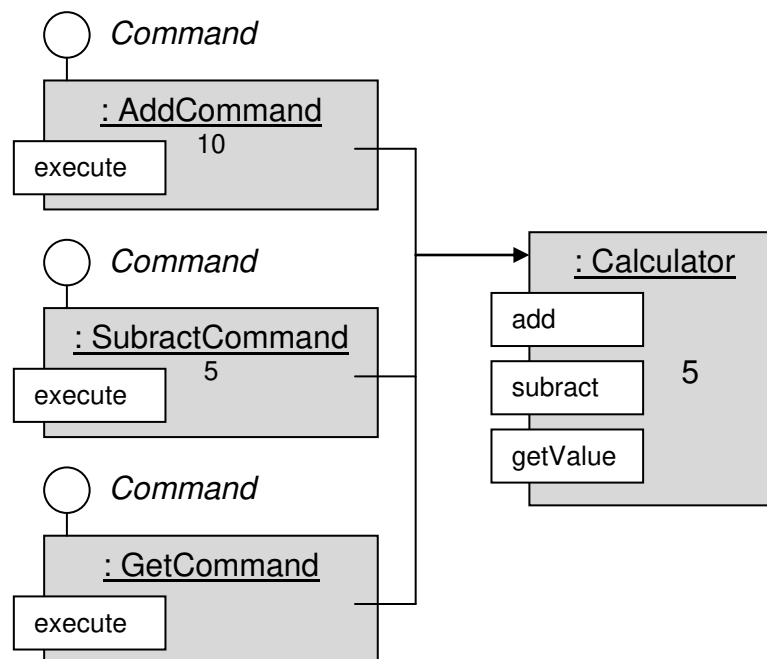
    private static int executeAddCommand(String[] tokens, int value) {
        if (tokens.length != 2) {
            System.out.println("Bad input");
            return value;
        }
        value += Integer.parseInt(tokens[1]);
        return value;
    }

    private static int executeSubtractCommand(String[] tokens, int value) {
        if (tokens.length != 2) {
            System.out.println("Bad input");
            return value;
        }
        value -= Integer.parseInt(tokens[1]);
    }
}
```

```
        return value;
    }
    private static int executeGetCommand(String[] tokens, int value) {
        if (tokens.length != 1) {
            System.out.println("Bad input");
            return value;
        }
        System.out.println(value);
        return value;
    }
}
```

Das obige Programm ist prozedural orientiert. Es wäre aber schöner, die Kommandos des Benutzers als Objekte zu modellieren – um sie z.B. in einer Historie sammeln zu können. Kommandos sollten dann auch die Fähigkeit haben, ihre Wirkung wieder rückgängig zu machen (undo/redo).

4.1.2 Lösung



Wir beginnen mit einem Interface:

```
package util;

public interface Command {
    public abstract void execute();
}
```

Ein Kommando muss sich ausführen können.

Ein Kommando benötigt einen Kontext, auf dem das Kommando ausgeführt wird. Im Falle des Taschenrechners:

```
package appl;

public class Calculator {
    private int value;
    public void add(int v) { this.value += v; }
    public void subtract(int v) { this.value -= v; }
    public int getValue() { return this.value; }
}
```

Dann kann für jede Kommando-Sorte eine eigene Klasse definiert werden, die jeweils `Command` implementiert:

```
package appl;

import util.Command;

public class AddCommand implements Command {
    private final Calculator calculator;
    private final int param;

    public AddCommand(Calculator calculator, String[] tokens) {
        if (tokens.length != 2)
            throw new RuntimeException("bad AddCommand");
        this.calculator = calculator;
        this.param = Integer.parseInt(tokens[1]);
    }

    @Override
    public void execute() {
        this.calculator.add(this.param);
    }
}
```

```
package appl;

import util.Command;

public class SubtractCommand implements Command {
    private final Calculator calculator;
    private final int param;

    public SubtractCommand(Calculator calculator, String[] tokens) {
        if (tokens.length != 2)
            throw new RuntimeException("bad SubtractCommand");
        this.calculator = calculator;
        this.param = Integer.parseInt(tokens[1]);
    }

    @Override
    public void execute() {
        this.calculator.subtract(this.param);
    }
}
```

```
package appl;

import util.Command;

public class GetCommand implements Command {
    private final Calculator calculator;

    public GetCommand(Calculator calculator, String[] tokens) {
        if (tokens.length != 1)
            throw new RuntimeException("bad GetCommand");
        this.calculator = calculator;
    }
}
```

```
@Override
public void execute() {
    System.out.println(this.calculator.getValue());
}
}
```

Dem Konstruktor jeder Kommando-Klasse wird die in Tokens zerlegte Eingabe übergeben (z.B.: "add", "20"). Der Konstruktor ist dann dafür zuständig, diese Tokens zu "parsen" und das Ergebnis dieses Parsens ggf. zu speichern (param). Dem Konstruktor wird weiterhin der Kontext übergeben (Calculator).

Die `execute`-Methode kann dann den `Calculator` nutzen und entsprechende Operationen auf ihn ausführen. Ein `AddCommand` ruft `add` auf, ein `SubtractCommand` die Methode `subtract` etc.

Hier die neue, "aufgeräumte" Anwendung:

```
package appl;
// ...
import util.Command;

public class Application {
    static void main(String[] args) throws Exception {
        final Calculator calculator = new Calculator();
        System.out.println("add n | subtract n | get | end : ");
        String input;
        final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        while (! (input = reader.readLine()).equals("end")) {
            input = input.trim();
            if (input.length() == 0)
                continue;
            final String[] tokens = input.split(" ");
            executeCommand(tokens, calculator);
        }
    }
    static void executeCommand(String[] tokens, Calculator calculator) {
        Command command = null;
        switch (tokens[0]) {
            case "add":
                command = new AddCommand(calculator, tokens);
                break;
            case "subtract":
                command = new SubtractCommand(calculator, tokens);
                break;
            case "get":
                command = new GetCommand(calculator, tokens);
                break;
            default:
                System.out.println("Bad command");
                break;
        }
        if (command != null)
            command.execute();
    }
}
```

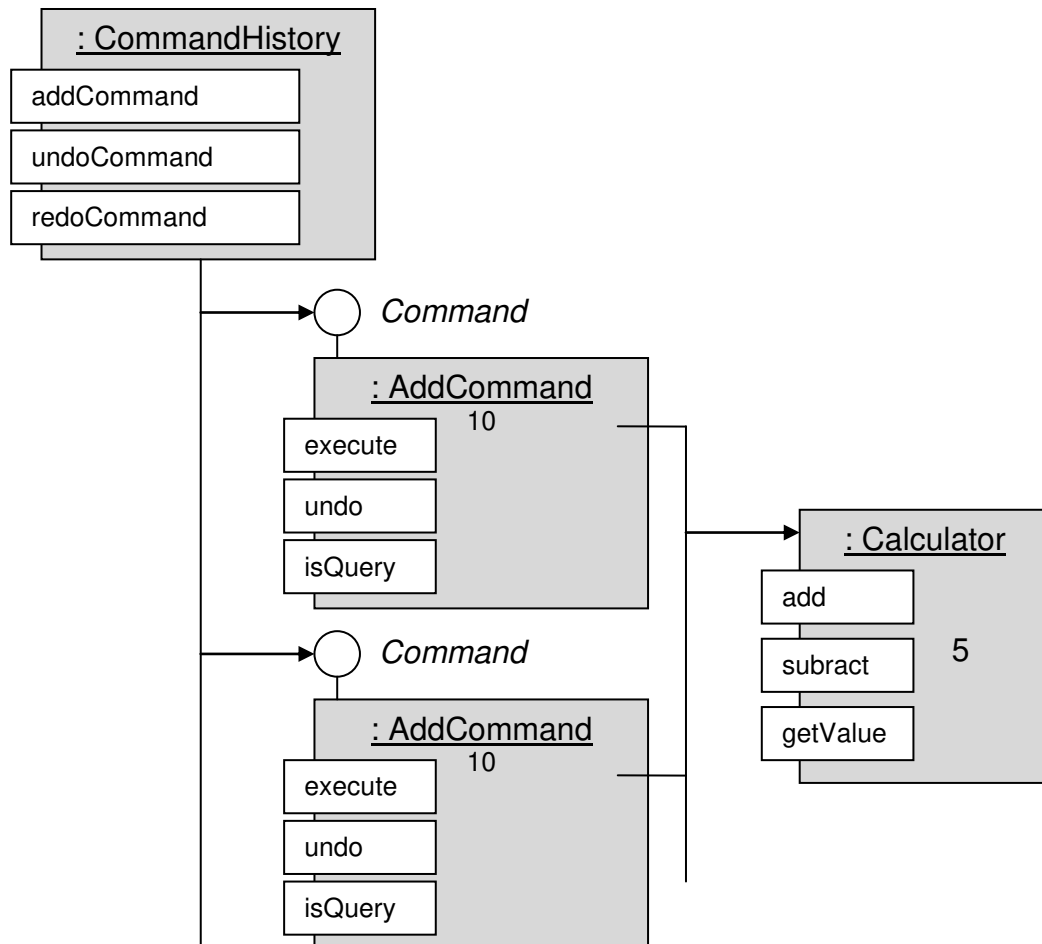

Die obige Anwendung zerlegt jede Zeile in einzelne Tokens und erzeugt abhängig vom Wert des ersten Tokens (dem "Kommandowort") ein entsprechendes Kommando-Objekt. Auf dieses Objekt wird dann die `Execute`-Methode aufgerufen. Und damit hat das Kommando-Objekt dann auch bereits seine Schuldigkeit getan.

Aufgaben

Ziehen Sie zwischen `Command` und `AddCommand` etc. eine weitere Schicht ein: `CalculatorCommand` (eine abstrakte Klasse). Warum ist das sinnvoll?

4.1.3 History

Natürlich kann man die Kommando-Objekte auch speichern und dann einen undo/redo-Mechanismus einführen.



Zunächst wird das Command-Interface erweitert:

```

package util;

public interface Command {
    public abstract void execute();
    public abstract boolean isQuery();
    public abstract void undo();
}

```

Ein Kommando sollte seine Wirkung rückgängig machen können. Und ein Kommando sollte Auskunft darüber geben können, ob es überhaupt eine Wirkung gehabt hat (ein `getCommand` hat keine Wirkung auf den Kontext). Liefert also `isQuery` den Wert `true`, so ist es unsinnig, `undo` aufzurufen.

Dementsprechend müssen die konkreten Kommando-Klassen erweitert werden:

```
package appl;

import util.Command;

public class AddCommand implements Command {
    private final Calculator calculator;
    private final int param;

    public AddCommand(Calculator calculator, String[] tokens) {
        if (tokens.length != 2)
            throw new RuntimeException("bad AddCommand");
        this.calculator = calculator;
        this.param = Integer.parseInt(tokens[1]);
    }

    @Override
    public void execute() { this.calculator.add(this.param); }
    @Override
    public boolean isQuery() { return false; }
    @Override
    public void undo() { this.calculator.subtract(this.param); }
}
```

```
package appl;

import util.Command;

public class SubtractCommand implements Command {
    private final Calculator calculator;
    private final int param;

    public SubtractCommand(Calculator calculator, String[] tokens) {
        if (tokens.length != 2)
            throw new RuntimeException("bad SubtractCommand");
        this.calculator = calculator;
        this.param = Integer.parseInt(tokens[1]);
    }

    @Override
    public void execute() { this.calculator.subtract(this.param); }
    @Override
    public boolean isQuery() { return false; }
    @Override
    public void undo() { this.calculator.add(this.param); }
}
```

```
package appl;

import util.Command;

public class GetCommand implements Command {
    private final Calculator calculator;

    public GetCommand(Calculator calculator, String[] tokens) {
```

```
        if (tokens.length != 1)
            throw new RuntimeException("bad GetCommand");
        this.calculator = calculator;
    }

    @Override
    public void execute() {
        System.out.println(this.calculator.getValue());
    }
    @Override
    public boolean isQuery() { return true; }
    @Override
    public void undo()          { throw new RuntimeException(); }
}
```

Dann kann eine `CommandHistory` eingeführt werden:

```
package util;
// ...
public class CommandHistory {

    private final List<Command> commands = new ArrayList<>();
    private int undoIndex = -1;

    public void addCommand(Command command) {
        for (int i = this.commands.size() - 1; i > this.undoIndex; i--)
            this.commands.remove(i);
        this.commands.add(command);
        this.undoIndex++;
    }

    public void undoCommand() {
        if (!this.canUndo())
            throw new RuntimeException("cannot undo");
        this.commands.get(this.undoIndex).undo();
        this.undoIndex--;
    }

    public void redoCommand() {
        if (!this.canRedo())
            throw new RuntimeException("cannot redo");
        this.undoIndex++;
        this.commands.get(this.undoIndex).execute();
    }

    public boolean canUndo() {
        return this.undoIndex >= 0;
    }

    public boolean canRedo() {
        return this.undoIndex < this.commands.size() - 1;
    }
}
```

`undoIndex` ist der Index desjenigen Kommandos, dessen Wirkung beim nächsten Aufruf von `undoCommand` rückgängig gemacht werden muss. `undoCommand` ermittelt das Kommando, dessen Wirkung rückgängig

gemacht werden muss, und ruft auf dieses Kommando dann die Methode `undo` auf. `redoCommand` führt ein `undo` auf ein vorhergehendes `undo` aus – indem einfach die `execute`-Methode des Kommandos erneut aufgerufen wird. Per `canUndo` resp. `canRedo` kann ermittelt werden, ob ein `undo` resp. `redo` im aktuellen Zustand möglich ist.

Die Anwendung schließlich kann dann wie folgt erweitert werden:

```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        final Calculator calculator = new Calculator();
        final CommandHistory history = new CommandHistory();
        System.out.println(
            "add n | subtract n | get | undo | redo | end : ");
        String input;
        final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        while (! (input = reader.readLine()).equals("end")) {
            input = input.trim();
            if (input.length() == 0)
                continue;
            final String[] tokens = input.split(" ");
            if (tokens[0].equals("undo")) {
                if (history.canUndo())
                    history.undoCommand();
                else
                    System.out.println("cannot undo");
            }
            else if (tokens[0].equals("redo")) {
                if (history.canRedo())
                    history.redoCommand();
                else
                    System.out.println("cannot redo");
            }
            else
                executeCommand(tokens, calculator, history);
        }
    }

    static void executeCommand(String[] tokens,
        Calculator calculator, CommandHistory history ) {
        Command command = null;
        switch (tokens[0]) {
            case "add":
                command = new AddCommand(calculator, tokens);
                break;
            case "subtract":
                command = new SubtractCommand(calculator, tokens);
                break;
            case "get":
                command = new GetCommand(calculator, tokens);
                break;
            default:
                System.out.println("Bad command");
                break;
        }
    }
}
```

```
        if (command != null) {  
            command.execute();  
            if (!command.isQuery())  
                history.addCommand(command);  
        }  
    }  
}
```

Man beachte, dass "undo" und "redo" keine Kommandos sind, sondern "Meta-Kommandos". Sie werden also nicht als `Command`-Objekte repräsentiert.

Ein beispielhafter Dialog (Ausgaben sind fett):

```
add n | subtract n | get | undo | redo | end :  
add 20  
subtract 5  
get  
15  
undo  
get  
20  
undo  
get  
0  
undo  
cannot undo  
redo  
redo  
get  
15  
redo  
cannot redo  
end
```

4.1.4 CommadFactory

Die Anwendung muss nun zwar nichts mehr über den Aufbau von Kommandos kennen (der Konstruktor der jeweiligen Kommando-Klasse ist für das "Parsen" der Kommandos zuständig), aber sie muss immer noch aufgrund des "Kommandowortes" ein passendes Kommando-Objekt erzeugen. Sie muss also immer dann angepasst werden, wenn die Liste der Kommando-Klassen um eine neue Klasse erweitert wird.

Angenommen, die Zuordnung zwischen den Kommandowörtern und den Kommando-Klassen ist in einer Textdatei hinterlegt ("Commands.properties"):

```
add      = appl.AddCommand
subtract = appl.SubtractCommand
get      = appl.GetCommand
```

Dann kann die Erzeugung der Kommando-Objekt auch "generisch" implementiert werden:

```
package util;
// ...
public class CommandFactory<C> {

    private final Map<String, Class<? extends Command>>
        commandClasses = new HashMap<>();

    public CommandFactory(String filename) throws Exception {
        final Properties props = new Properties();
        props.load(new FileInputStream(filename));
        for (final Map.Entry<Object, Object> entry : props.entrySet()) {
            final String className = (String) entry.getValue();
            final String op = (String) entry.getKey();
            final Class<? extends Command> cls =
                (Class<? extends Command>)Class.forName(className);
            this.commandClasses.put(op, cls);
        }
    }

    public Command createCommand(C context, String[] tokens)
        throws Exception {
        final String commandWord = tokens[0];
        final Class<? extends Command> cls =
            this.commandClasses.get(commandWord);
        if (cls == null)
            return null;
        final Constructor ctor = cls.getConstructor(
            context.getClass(), String[].class);
        return (Command) ctor.newInstance(context, tokens);
    }
}
```

Man beachte die Parametrisierung der `CommandFactory`-Klasse mit einem Typ-Parameter (c).

Hier die geänderte Anwendung:

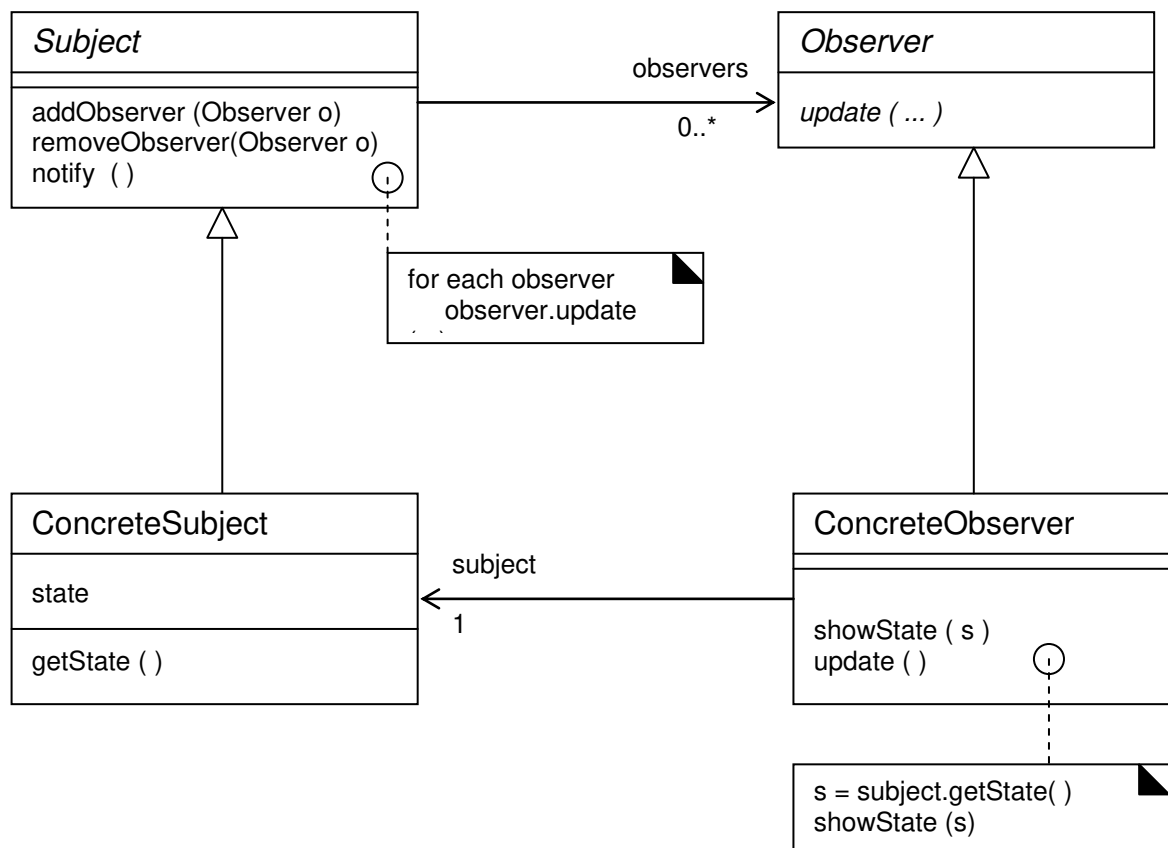
```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        final CommandFactory<Calculator> factory =
            new CommandFactory<>("src/commands.properties");
        final Calculator calculator = new Calculator();
        final CommandHistory history = new CommandHistory();
        System.out.println(
            "add n | subtract n | get | undo | redo | end : ");
        String input;
        final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        while (!(input = reader.readLine()).equals("end")) {
            input = input.trim();
            if (input.length() == 0)
                continue;
            final String[] tokens = input.split(" ");
            if (tokens[0].equals("undo")) {
                if (history.canUndo())
                    history.undoCommand();
                else
                    System.out.println("cannot undo");
            }
            else if (tokens[0].equals("redo")) {
                if (history.canRedo())
                    history.redoCommand();
                else
                    System.out.println("cannot redo");
            }
            else {
                final Command command =
                    factory.createCommand(calculator, tokens);
                if (command == null)
                    System.out.println("Unknown Command Word");
                else {
                    command.execute();
                    if (!command.isQuery())
                        history.addCommand(command);
                }
            }
        }
    }
}
```

Als Context der CommandFactory wird hier die Klasse Calculator benutzt.

4.2 Observer

"Definiere eine 1-zu-n Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden." (Gamma, 257)



4.2.1 Problem

Ein Taschenrechner soll bei seiner Arbeit beobachtet werden.

Hier die Calculator-Klasse:

```
package appl;

public class Calculator {
    private int value;
    public void add(int v)      { this.value += v; }
    public void subtract(int v) { this.value -= v; }
    public int getValue()      { return this.value; }
}
```

Die Operationen, die auf einen Calculator ausgeführt werden, sollen in einer Datenbank protokolliert werden und auf einem Display ausgegeben werden.

Hier die Datenbank:

```
package appl;

class Database {
    public void insert(Object message) {
        System.out.println("Database: " + message);
    }
}
```

Und hier das Display:

```
package appl;

class Display {
    public void write(Object message) {
        System.out.println("Display: " + message);
    }
}
```

Hier eine kleine Applikation:

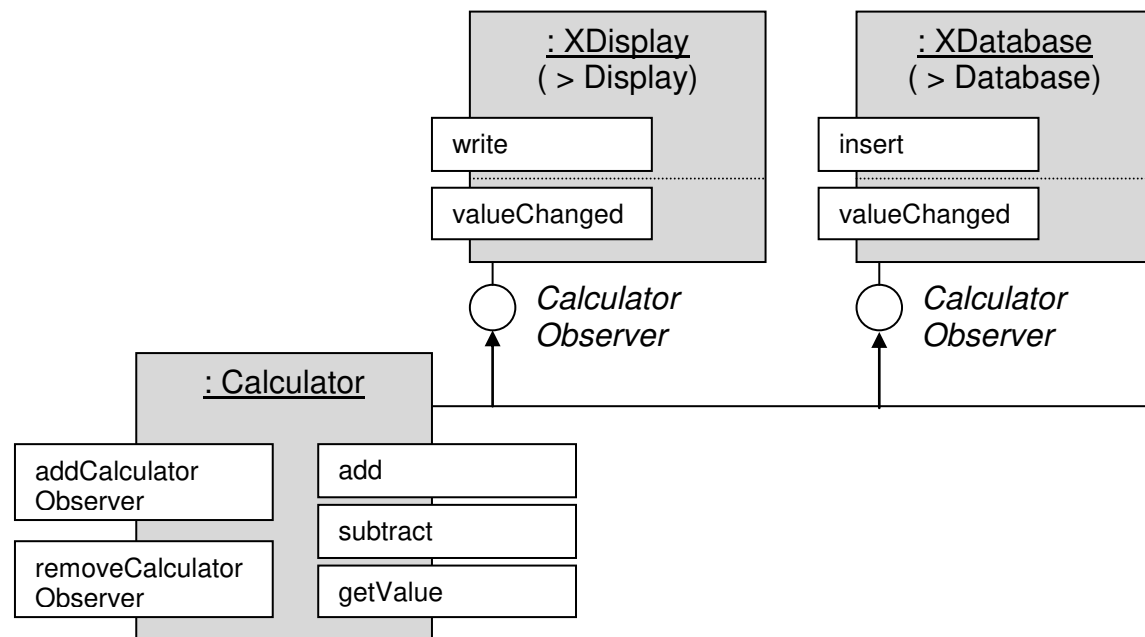
```
package appl;

public class Application {

    public static void main(String[] args) throws Exception {
        final Calculator calculator = new Calculator();
        final Display display = new Display();
        final Database database = new Database();
        calculator.add(20);
        display.write("add 20");
        database.insert("add 20");
        calculator.subtract(5);
        display.write("subtract 5");
        database.insert("subtract 5");
        System.out.println("value = " + calculator.getValue());
    }
}
```

Jedes Mal dann, wenn eine Operation auf dem `Calculator` ausgeführt wurde, liegt es in der Verantwortung der `Application`, die Operation zu protokollieren und anzuzeigen. Es wäre besser, wenn die Protokollierung und die Anzeige "automatisch" ausgeführt würden.

4.2.2 Lösung



Zunächst wird ein Interface definiert:

```

package appl;

public interface CalculatorObserver {
    public abstract void valueChanged(CalculatorEvent e);
}
  
```

Ein `CalculatorObserver` wird einen `Calculator` beobachten. Immer dann, wenn eine `Calculator`-Operation ausgeführt wird, wird die `valueChanged`-Methode eines solchen Observers aufgerufen. Dieser Methode wird ein `CalculatorEvent` übergeben – ein Objekt, welches nähere Einzelheiten über die Art der ausgeführten Operation, über die beteiligten Parameter etc. enthält:

```

package appl;

public class CalculatorEvent {
    public enum Type {
        ADD, SUBTRACT
    };

    private final Calculator source;
    private final Type type;
    private final int value;

    public CalculatorEvent(Calculator source, Type type, int value) {
        this.source = source;
        this.type = type;
        this.value = value;
    }
}
  
```

```

public Calculator getSource() { return this.source; }
public Type getType()         { return this.type; }
public int getValue()         { return this.value; }

@Override
public String toString()      { ... }
}

```

Man beachte, dass `CalculatorEvent`-Objekte konstant sind.

Dann werden zwei Klassenadapter gebaut (siehe Adapter-Muster):

```

package appl;

public class XDatabase extends Database implements CalculatorObserver {
    @Override
    public void valueChanged(CalculatorEvent e) {
        this.insert("XDatabase: " + e);
    }
}

```

```

package appl;

public class XDisplay extends Display implements CalculatorObserver {
    @Override
    public void valueChanged(CalculatorEvent e) {
        this.write("XDisplay: " + e);
    }
}

```

Sowohl `XDatabase`- als auch `XDisplay`-Objekte können nun als `ICalculatorObserver` fungieren.

Schließlich wird die Klasse `Calculator` erweitert:

```

package appl;
// ...
public class Calculator {
    private final List<CalculatorObserver> observers = new
ArrayList<>();
    public void addCalculatorObserver(CalculatorObserver observer) {
        this.observers.add(observer);
    }
    public void removeCalculatorObserver(CalculatorObserver observer) {
        this.observers.remove(observer);
    }
    private int value;
    public void add(int v) {
        this.value += v;
        this.fireValueChanged(CalculatorEvent.Type.ADD, v);
    }
    public void subtract(int v) {
        this.value -= v;
        this.fireValueChanged(CalculatorEvent.Type.SUBTRACT, v);
    }
    public int getValue() {
        return this.value;
    }
}

```

```
}  
private void fireValueChanged(CalculatorEvent.Type type, int v) {  
    final CalculatorEvent e = new CalculatorEvent(this, type, v);  
    this.observers.forEach(observer -> observer.valueChanged(e));  
}  
}
```

Bei einem `Calculator` können nun `CalculatorObserver` registriert werden (per `addCalculatorListener`) – und können auch wieder de-registriert werden (per `removeCalculatorListener`).

Immer dann, wenn `add` oder `subtract` aufgerufen wird, werden nach der entsprechenden Zustandsänderung des `Calculators` alle registrierten Listener (Observer) benachrichtigt – über den Aufruf ihrer jeweiligen `valueChanged`-Methode. Dies ist die Aufgabe von `fireValueChanged`, einer Hilfsmethode, an welche `add` und `subtract` delegieren.

Die Anwendung muss dann nur noch die Registrierung der Observer vornehmen – und kann sich anschließend ausschließlich auf den `Calculator` konzentrieren:

```
package appl;  
  
public class Application {  
    public static void main(String[] args) throws Exception {  
        final XDisplay display = new XDisplay();  
        final XDatabase database = new XDatabase();  
        final Calculator calculator = new Calculator();  
        calculator.addCalculatorObserver(display);  
        calculator.addCalculatorObserver(database);  
        calculator.add(20);  
        calculator.subtract(5);  
        System.out.println("value = " + calculator.getValue());  
    }  
}
```

Die Ausgabe des Programms:

```
Display: XDisplay: ADD 20  
Database: XDatabase: ADD 20  
Display: XDisplay: SUBTRACT 5  
Database: XDatabase: SUBTRACT 5  
value = 15
```

4.2.3 Lambdas

Man kann das Observer-Pattern auch mit Lambdas implementieren.

Statt des `CalculatorObserver`-Interfaces wird das funktionale Standard-Interface `Consumer` verwendet.

Ein Handler, welcher über `CalculatorEvents` benachrichtigt werden soll, benötigt eine `void`-Methode (beliebigen Namens), welche exakt einen einzigen Parameter des Typs `CalculatorEvent` besitzt.

Die Klassen `XDatabase` und `XDisplay` müssen dann jeweils eine solche Handler-Methode bereitstellen:

```
package appl;

public class XDatabase extends Database {
    public void insert(CalculatorEvent e) {
        this.insert("XDatabase: " + e);
    }
}
```

```
package appl;

public class XDisplay extends Display {
    public void write(CalculatorEvent e) {
        this.write("XDisplay: " + e);
    }
}
```

Die `Calculator`-Klasse wird wie folgt geändert:

```
package appl;
// ...
import java.util.function.Consumer;

public class Calculator {

    private final List<Consumer<CalculatorEvent>> observers =
        new ArrayList<>();

    public void addCalculatorObserver(
        Consumer<CalculatorEvent> observer) {
        this.observers.add(observer);
    }
    public void removeCalculatorObserver(
        Consumer<CalculatorEvent> observer) {
        this.observers.remove(observer);
    }

    private int value;
    public void add(int v) {
        this.value += v;
        this.fireValueChanged(CalculatorEvent.Type.ADD, v);
    }
    public void subtract(int v) {
        this.value -= v;
        this.fireValueChanged(CalculatorEvent.Type.SUBTRACT, v);
    }
}
```

```
}  
public int getValue() {  
    return this.value;  
}  
private void fireValueChanged(CalculatorEvent.Type type, int v) {  
    final CalculatorEvent e = new CalculatorEvent(this, type, v);  
    this.observers.forEach(observer -> observer.accept(e));  
}  
}
```

Die Anwendung sieht dann wie folgt aus:

```
package appl;  
  
public class Application {  
  
    public static void main(String[] args) throws Exception {  
        final XDisplay display = new XDisplay();  
        final XDatabase database = new XDatabase();  
        final Calculator calculator = new Calculator();  
        calculator.addCalculatorObserver(e -> display.write(e));  
        calculator.addCalculatorObserver(e -> database.insert(e));  
        calculator.add(20);  
        calculator.subtract(5);  
        System.out.println("value = " + calculator.getValue());  
    }  
}
```

Den Aufrufen von `addCalculatorEventHandler` werden nun Consumer übergeben, welche `CalculatorEvents` konsumieren können.

Aufgaben

Welche Vorteile resp. Nachteile sehen Sie in dieser Variante – verglichen mit der Interface-basierten Variante?

4.2.4 Vereinfachung

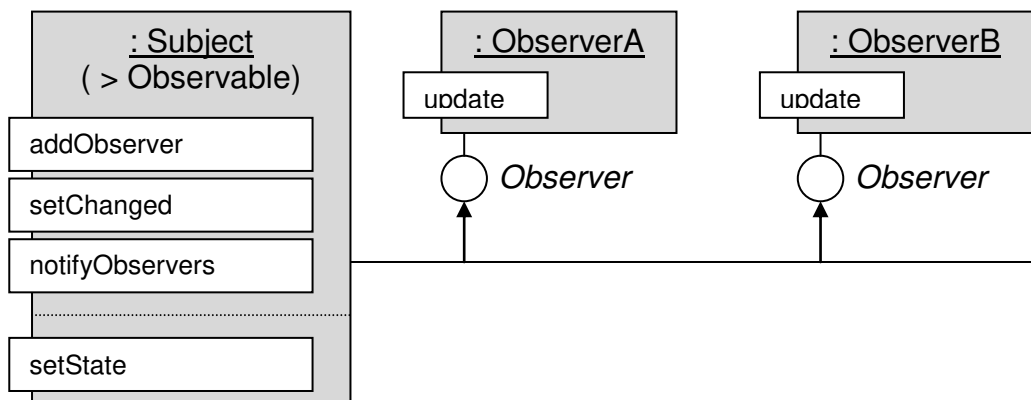
Natürlich kommt man nun auch ganz ohne die Klasse `XDisplay` und `XDatabase` aus. Hier also eine noch einfachere Lösung:

```
package appl;

public class Application {

    public static void main(String[] args) throws Exception {
        final Display display = new Display();
        final Database database = new Database();
        final Calculator calculator = new Calculator();
        calculator.addCalculatorObserver(e -> display.write(e));
        calculator.addCalculatorObserver(e -> database.insert(e));
        calculator.add(20);
        calculator.subtract(5);
        System.out.println("value = " + calculator.getValue());
    }
}
```

4.2.5 Observer/Observable



Die Standardbibliothek von Java enthält das Interface `Observer` und die abstrakte Basisklasse `Observable`. Das folgende Beispiel demonstriert die Verwendung dieser Typen.

Objekte der folgende Klasse `Subject` sind `Observable`:

```

package appl;

import java.util.Observable;

public class Subject extends Observable {

    private int state;

    public void setState(int state) {
        this.state = state;
        this.setChanged();
        this.notifyObservers();
    }

    public int getState() {
        return this.state;
    }
}
  
```

`Observable` definiert die Methoden `setChanged` und `notifyObservers`. Diese Basisklasse definiert eine Liste, in welcher Objekte eingetragen werden können, deren Klassen das Interface `Observer` implementieren. `notifyObservers` benachrichtigt alle in dieser Liste registrierten `Observer` – sofern zuvor `setChanged` aufgerufen wurde.

Hier zwei Klassen, die dieses Interface implementieren:

```
package appl;

import java.util.Observer;
import java.util.Observable;

public class ObserverA implements Observer {

    @Override
    public void update(Observable o, Object arg) {
        Subject subject = (Subject) o;
        System.out.println(
            "ObserverA: new State = " + subject.getState());
    }
}
```

```
package appl;

import java.util.Observable;
import java.util.Observer;

public class ObserverB implements Observer {

    @Override
    public void update(Observable o, Object arg) {
        final Subject subject = (Subject) o;
        System.out.println(
            "ObserverB: new State = " + subject.getState());
    }
}
```

Die folgende Anwendung erzeugt ein Subject und registriert bei diesem Subject jeweils eine Instanz von ObserverA und ObserverB. Anschließend wird der Zustand des Subjects zweimal geändert:

```
package appl;

public class Application {

    public static void main(String[] args) {
        Subject subject = new Subject();
        subject.addObserver(new ObserverA());
        subject.addObserver(new ObserverB());
        subject.setState(333);
        subject.setState(4711);
    }
}
```

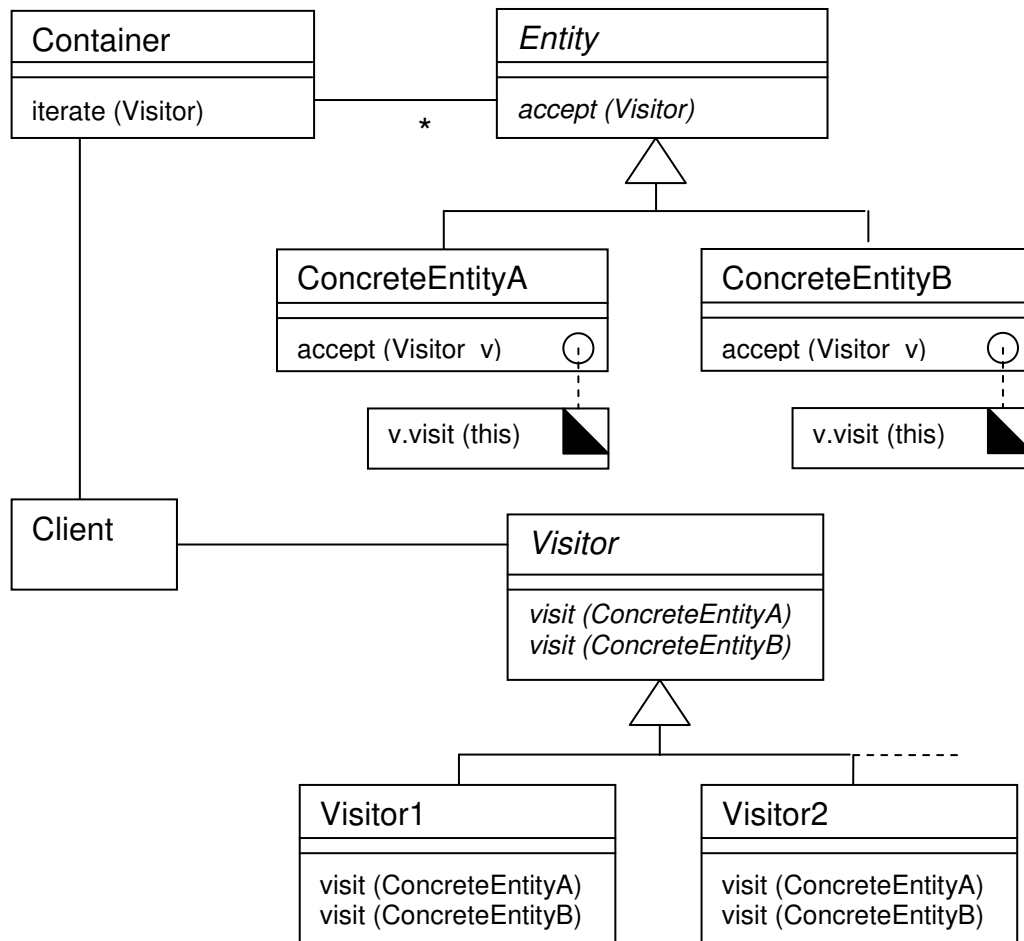
Die Ausgaben:

```
ObserverB: new State = 333  
ObserverA: new State = 333  
ObserverB: new State = 4711  
ObserverA: new State = 4711
```

Der Nachteil dieser Standard-Typen liegt darin, dass fachliche Klassen (und `Subject` steht für irgendeine solche Klasse) abhängig werden von einer technischen Basisklasse (von `Observable`).

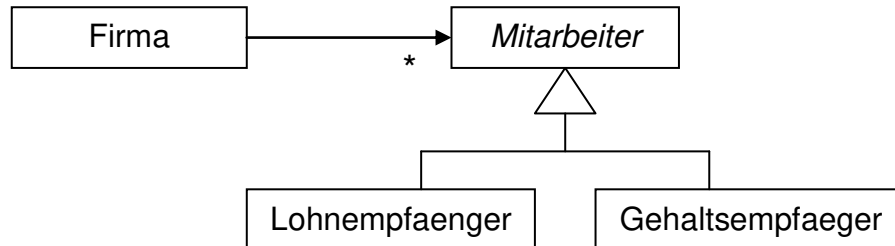
4.3 Visitor

"Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es Ihnen, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern." (Gamma, 269)



4.3.1 Problem

Gegeben sei folgende Hierarchie von Mitarbeiter-Klassen:



```
package appl;

public abstract class Mitarbeiter {
    public final int nr;
    public final String name;

    public Mitarbeiter(int nr, String name) {
        this.nr = nr;
        this.name = name;
    }
    abstract public double getVerdienst();
    @Override
    public String toString() {
        return this.nr + ", " + this.name;
    }
}
```

```
package appl;

public class Lohnempfaenger extends Mitarbeiter {
    public final double anzStd;
    public final double stdLohn;

    public Lohnempfaenger(int nr, String name,
        double anzStd, double stdLohn) {
        super(nr, name);
        this.anzStd = anzStd;
        this.stdLohn = stdLohn;
    }
    @Override
    public double getVerdienst() {
        return this.anzStd * this.stdLohn;
    }
    @Override
    public String toString() {
        return super.toString() + ", " +
            this.anzStd + ", " + this.stdLohn;
    }
}
```

```
package appl;

public class Gehaltsempfaenger extends Mitarbeiter {
    public final double gehalt;

    public Gehaltsempfaenger(int nr, String name, double gehalt) {
        super(nr, name);
        this.gehalt = gehalt;
    }
    @Override
    public double getVerdienst() {
        return this.gehalt;
    }
    @Override
    public String toString() {
        return super.toString() + ", " + this.gehalt;
    }
}
```

Eine Firma dient als Container für Mitarbeiter:

```
package appl;
// ...
public class Firma {
    private final List<Mitarbeiter> list = new ArrayList<>();

    public void add(Mitarbeiter mitarbeiter) {
        this.list.add(mitarbeiter);
    }
    public List<Mitarbeiter> getList() {
        return Collections.unmodifiableList(this.list);
    }
    public double getGesamtVerdienst() {
        double summe = 0;
        for (final Mitarbeiter m : this.list)
            summe += m.getVerdienst();
        return summe;
    }
}
```

Eine Anwendung möchte nun den Gesamtverdienst aller Mitarbeiter ausgeben, die kompletten Daten aller Mitarbeiter und den Gesamtverdienst aller Gehaltsempfänger. Die erste Aufgabe ist problemlos: Mitarbeiter definiert eine abstrakte Methode `getVerdienst`, die in der `getGesamtVerdienst`-Methode der Firma aufgerufen wird.

Die zweite und dritte Aufgabe sind problematisch:

```
package appl;

public class Application {
    public static void main(String[] args) throws Exception {
        final Firma f = new Firma();

        f.add(new Lohnempfaenger(1000, "Meier", 160, 25));
        f.add(new Gehaltsempfaenger(2000, "Mueller", 3000));
        f.add(new Lohnempfaenger(3000, "Franke", 80, 25));

        System.out.println(
            "Gesamtverdienst = " + f.getGesamtVerdienst());

        for (final Mitarbeiter m : f.getList()) {
            System.out.print(m.nr + " " + m.name + " ");
            if (m instanceof Lohnempfaenger) {
                final Lohnempfaenger l = (Lohnempfaenger) m;
                System.out.println(l.anzStd + " " + l.stdLohn);
            }
            else if (m instanceof Gehaltsempfaenger) {
                final Gehaltsempfaenger g = (Gehaltsempfaenger) m;
                System.out.println(g.gehalt);
            }
        }

        double summe = 0;
        for (final Mitarbeiter m : f.getList()) {
            if (m instanceof Gehaltsempfaenger)
                summe += m.getVerdienst();
        }
        System.out.println(
            "Gesamtverdienst aller Gehaltsempfaenger = " + summe);
    }
}
```

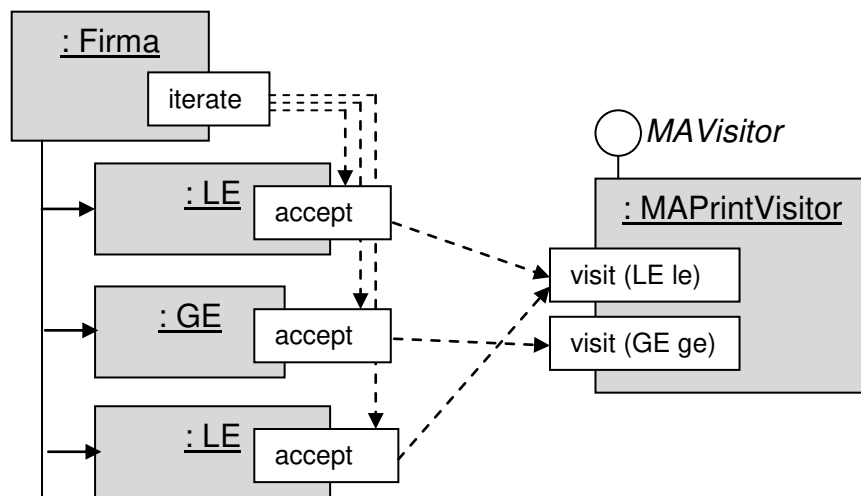
In beiden Fällen (Ausgabe aller Mitarbeiterdaten und Berechnung des Verdienstes aller Gehaltsempfänger) ist eine dynamische Typabfrage erforderlich – und der damit zusammenhängende Downcast (die objektorientierte "Brechtange": und wenn man nicht mehr weiter kann, dann wendet man den Downcast an).

Es wäre schön, wenn man auf diese beiden "schmutzigen" Elemente verzichten könnte.

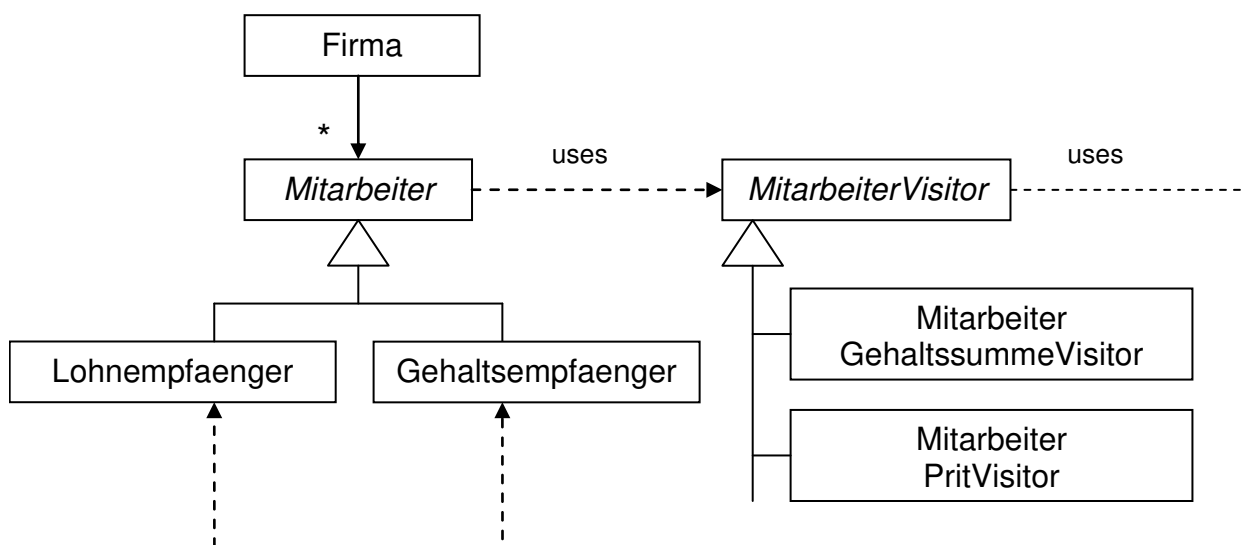
Man könnte z.B. in `Mitarbeiter` bereits eine `print`-Methode vorsehen, die in `Lohnempfaenger` und `Gehaltsempfaenger` überschrieben wird – dann könnte auch hier der Mechanismus des dynamischen Bindens verwendet werden. Und man könnte ebenso eine Methode `getGehaltsempfaengerVerdienst` vorsehen, die in `Mitarbeiter` den Wert 0 liefert und in `Gehaltsempfaenger` überschrieben wird – und könnte somit auch hier das dynamische Binden nutzen. Auf diese Weise erhält man aber leicht eine Inflation neuer Methoden, welche die zentralen Klassen (`Mitarbeiter`, `Gehaltsempfaenger`, `Lohnempfaenger`) in unerwünschter Weise aufblähen. Denn viele solcher Methoden werden einfach aus einem ad-hoc-Bedürfnis entspringen – und somit der eigentlichen Sache ganz äußerlich bleiben (heute soll so gedruckt werden, morgen wieder ganz anders...).

4.3.2 Lösung

Ein Objektdiagramm:



Das Klassendiagramm:



Wir beginnen mit einem Interface:

```

package appl;

public interface MitarbeiterVisitor {
    public abstract void visit(Lohnempfaenger l);
    public abstract void visit(Gehaltsempfaenger g);
}
  
```

Das Interface enthält für jeden instanziiierbaren Mitarbeiter-Typ (also für Lohnempfaenger und Gehaltsempfaenger) eine überladene visit-Methode.

Ein `MitarbeiterVisitor` (ein Objekt einer dieses Interface implementierenden Klasse) kann einen `Lohnempfaenger` und einen `Gehaltsempfaenger` besuchen: zu diesem Zweck wird eine der beiden `visit`-Methoden aufgerufen werden.

Aufgrund dieses Interfaces kann nun eine `MitarbeiterPrintVisitor`-Klasse definiert werden:

```
package appl;

public class MitarbeiterPrintVisitor implements MitarbeiterVisitor {
    @Override
    public void visit(Lohnempfaenger l) {
        System.out.println(l.nr + " " + l.name + " " +
            l.stdLohn + " " + l.anzStd);
    }
    @Override
    public void visit(Gehaltsempfaenger g) {
        System.out.println(g.nr + " " + g.name + " " +
            g.gehalt);
    }
}
```

Und eine Klasse, welche den Gesamtverdienst aller Gehaltsempfänger berechnet:

```
package appl;

public class MitarbeiterGehaltssummeVisitor implements MitarbeiterVisitor {
    private int summe = 0;
    @Override
    public void visit(Lohnempfaenger l) {
        // do nothing
    }
    @Override
    public void visit(Gehaltsempfaenger g) {
        this.summe += g.getVerdienst();
    }
    public int getSumme() {
        return this.summe;
    }
}
```

Dann muss es nur noch ein Verfahren geben, einen solchen Visitor an einen `Gehaltsempfaenger` oder `Lohnempfaenger` heranzuführen – irgendjemand muss die entsprechende `visit`-Methode aufrufen.

Und das überlässt man am besten den `Mitarbeiter`-Klassen selbst.

Die Basisklasse `Mitarbeiter` wird um eine abstrakte Methode `accept` erweitert:

```
public abstract class Mitarbeiter {
    // ...
    abstract public void accept(MitarbeiterVisitor visitor);
}
```

Die `accept`-Methode ist mit jedem `MitarbeiterVisitor`-kompatiblen Objekt zufrieden.

Diese Methode muss dann natürlich in den abgeleiteten Klassen implementiert werden:

```
public class Lohnempfaenger extends Mitarbeiter {
    // ...
    @Override
    public void accept(MitarbeiterVisitor visitor) {
        visitor.visit(this);
    }
}
```

```
public class Gehaltsempfaenger extends Mitarbeiter {
    // ...
    @Override
    public void accept(MitarbeiterVisitor visitor) {
        visitor.visit(this);
    }
}
```

In beiden Fällen handelt es sich vordergründig um dieselbe Implementierung (textuell sind die Implementierungen identisch) – tatsächlich aber unterscheiden sich die beiden Implementierungen. Denn `this` ist im ersten Falle vom statischen Typ `Lohnempfaenger`, und im zweiten Falle vom statischen Typ `Gehaltsempfaenger`. Im ersten Falle wird somit die `visit(Lohnempfaenger)`-Methode des übergebenen Visitors aufgerufen, im zweiten Falle dessen `visit(Gehaltsempfaenger)`-Methode.

Eben wegen der unterschiedlichen Implementierung der `accept`-Methoden kann es auch keine `accept`-Methode in der Basisklasse `MA` geben (dort wäre `this` vom Typ `Mitarbeiter` – und eine `visit`-Methode mit einem `Mitarbeiter`-Parameter gibt's nicht).

Schließlich wird die Klasse `Firma` um eine `iterate`-Methode erweitert:

```
public class Firma {
    private final List<Mitarbeiter> list = new ArrayList<>();
    // ...
    public void iterate(MitarbeiterVisitor visitor) {
        for (final Mitarbeiter m : this.list)
            m.accept(visitor);
    }
}
```

Die Anwendung des letzten Abschnitts kann dann wie folgt modifiziert werden:

```
package appl;

public class Application {

    public static void main(String[] args) throws Exception {
        final Firma f = new Firma();

        f.add(new Lohnempfaenger(1000, "Meier", 160, 25));
        f.add(new Gehaltsempfaenger(2000, "Mueller", 3000));
        f.add(new Lohnempfaenger(3000, "Franke", 80, 25));

        System.out.println("Gesamtverdienst = " +
            f.getGesamtVerdienst());
        f.iterate(new MitarbeiterPrintVisitor());
        final MitarbeiterGehaltssummeVisitor v =
            new MitarbeiterGehaltssummeVisitor();
        f.iterate(v);
        System.out.println(
            "Gesamtverdienst aller Gehaltsempfaenger = " +
            v.getSumme());
    }
}
```

Leider ist das Interface `MitarbeiterVisitor` abhängig von der konkreten `Mitarbeiter`-Klassenhierarchie – also von `Lohnempfaenger` und `Gehaltsempfaenger`. Sollte diese Klassenhierarchie irgendwann einmal um eine Klasse erweitert werden, muss auch das Interface `MitarbeiterVisitor` um eine neue `visit`-Methode erweitert werden – mit allen Konsequenzen, die sich daraus für die Implementierungen des Interfaces ergeben.

Das Visitor-Pattern kann man also in der reinen, von Gamma beschriebenen Form guten Gewissens nur dann verwenden, wenn die Hierarchie der Klassen, deren Objekte von den Besuchern besucht werden, "endgültig" ist. Leider ist das wahrscheinlich nur selten der Fall...

Aber das Muster bietet auf jeden Fall den Vorteil, ad-hoc-Funktionalität von den eigentlichen Klassen fernhalten zu können.

Aufgaben

Analysieren Sie die Fehlermeldung des Compilers, wenn versucht wird, die `accept`-Methode bereits in der Basisklasse (`Mitarbeiter`) zu implementieren.

Ziehen Sie zwischen dem Interface `MitarbeiterVisitor` und den Implementierungsklassen eine weitere Schicht ein: `AbstractMitarbeiterVisitor`. Worin besteht der Sinn einer solchen Klasse?

4.3.3 Dispatch mit Reflection

Im folgenden geht's um eine Erweiterung des Visitor-Patterns, welche die Vorteile des Patterns beibehält, dessen Nachteile aber vermeidet.

Die erste Überlegung muss dann darin bestehen, auf das Interface `MitarbeiterVisitor` ganz zu verzichten – denn die Problematik des Visitor-Patterns besteht ja gerade darin, dass dieses Interface immer dann erweitert werden muss, wenn neue `MA`-Klassen hinzukommen.

Dann sehen die Visitor-Klassen wie folgt aus:

```
package appl;

public class MitarbeiterPrintVisitor {
    public void visit(Lohnempfaenger l) {
        System.out.println(l.nr + " " + l.name + " " +
            l.stdLohn + " " + l.anzStd);
    }
    public void visit(Gehaltsempfaenger g) {
        System.out.println(g.nr + " " + g.name + " " +
            g.gehalt);
    }
}
```

```
package appl;

public class MitarbeiterGehaltssummeVisitor {
    private int summe = 0;
    public void visit(Gehaltsempfaenger g) {
        this.summe += g.getVerdienst();
    }
    public int getSumme() {
        return this.summe;
    }
}
```

Die Visitor-Klassen sind nun nur noch "Visitor"-Klassen in Anführungsstrichen...

Man beachte, dass `MitarbeiterGehaltssummeVisitor` keine `visit`-Methode mehr besitzt, die einen `Lohnempfaenger` Parameter hat.!

Die Visitor-Klassen sind nur noch von `Object` abgeleitet. Per Vereinbarung legen wir aber fest, dass eine Klasse, deren Objekte als Besucher fungieren, für jede Klasse, deren Objekte zu besuchen sind, eine entsprechend parametrisierte Methode namens `visit` besitzt (eine Vereinbarung, deren Einhaltung vom Compiler natürlich nicht garantiert werden kann).

Weiterhin werden die `accept`-Methoden aus der `Mitarbeiter`-Klassen und der von ihr abgeleiteten Klassen ersatzlos gestrichen.

Die `iterate`-Methode der Klasse `Firma` schließlich wird wie folgt geändert:

```
package appl;
// ...
import util.Dispatcher;

public class Firma {
    private final List<Mitarbeiter> list = new ArrayList<>();

    public void add(Mitarbeiter mitarbeiter) { ... }
    public List<Mitarbeiter> getList() { ... }
    public double getGesamtVerdienst() { ... }

    public void iterate(Object visitor) {
        for (final Mitarbeiter m : this.list)
            Dispatcher.dispatch(visitor, "visit", m);
    }
}
```

Die Klasse benutzt nun eine Utility-Klasse: die Klasse `Dispatcher` (s. weiter unten).

Jeder `Mitarbeiter` wird an die `dispatch`-Methode der `Dispatcher`-Klasse übergeben. Zusätzlich wird der Name der `visit`-Methode übergeben ("visit") und der aufzurufende "Visitor".

Hier die Klasse `Dispatcher`:

```
package util;

import java.lang.reflect.Method;

public class Dispatcher {
    public static void dispatch(
        Object target, String methodName, Object parameter) {
        final Class<?> targetClass = target.getClass();
        final Class<?> parameterClass = parameter.getClass();
        try {
            final Method method =
                targetClass.getMethod(methodName, parameterClass);
            method.invoke(target, parameter);
        }
        catch (final Exception e) {
            // do nothing...
        }
    }
}
```

Die Analyse dieser Klasse sein dem Leser / der Leserin überlassen...

Wir könnten noch eine etwas intelligentere Version dieser Dispatcher-Klasse schreiben:

```
package util;

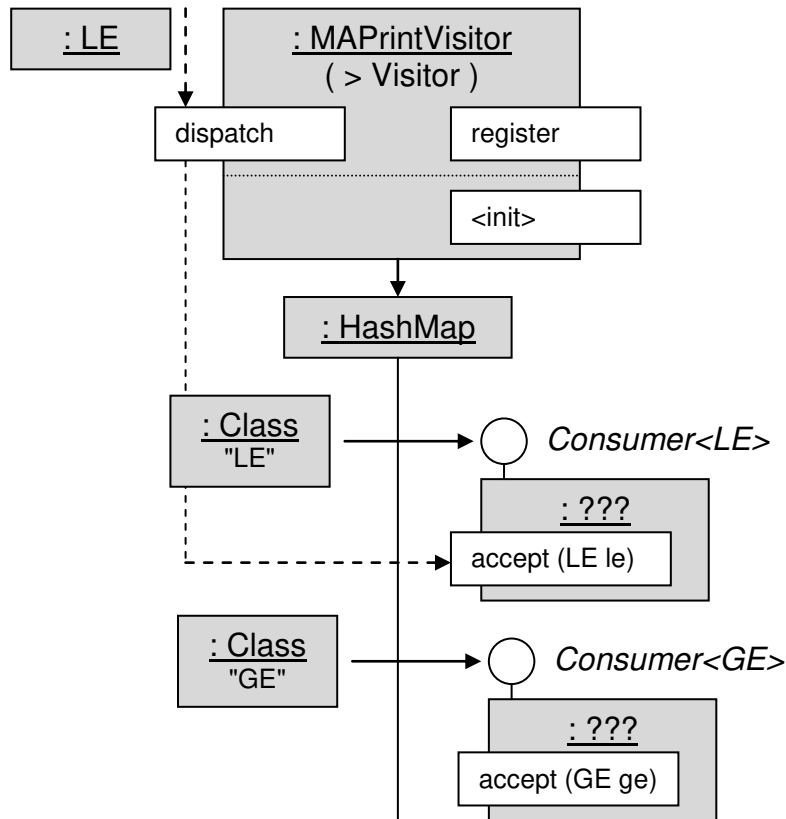
import java.lang.reflect.Method;

public class Dispatcher {
    public static void dispatch(
        Object target, String methodName, Object parameter) {
        final Class<?> targetClass = target.getClass();
        Class<?> parameterClass = parameter.getClass();
        while (parameterClass != null) {
            try {
                final Method method =
                    targetClass.getMethod(methodName, parameterClass);
                method.invoke(target, parameter);
                return;
            }
            catch (final Exception e) {
                parameterClass = parameterClass.getSuperclass();
            }
        }
    }
}
```

Worin liegt die zusätzliche Intelligenz dieser Klasse?

4.3.4 Dispatch mit Consumer

Die letzte Variante operierte Reflection-basiert. Wir können aber auch eine nicht-Reflection-basierte Lösung benutzen.



Wir definieren folgende Utility-Klasse:

```

package util;
// ...
import java.util.function.Consumer;

public abstract class Visitor {

    private final Map<Class<?>, Consumer<?>> handlers = new HashMap<>();

    public <T> void register(Class<T> cls, Consumer<T> handler) {
        this.handlers.put(cls, handler);
    }

    @SuppressWarnings({ "unchecked", "rawtypes" })
    public void dispatch(Object parameter) {
        Class<?> parameterClass = parameter.getClass();
        while (parameterClass != null) {
            final Consumer handler = this.handlers.get(parameterClass);
            if (handler != null) {
                handler.accept(parameter);
                return;
            }
            parameterClass = parameterClass.getSuperclass();
        }
    }
}

```

Diese Utility-Klasse wird nun als Basisklasse für die konkreten Visitor-Klassen genutzt:

```
package appl;

import java.util.function.Consumer;

import util.Visitor;

public class MitarbeiterPrintVisitor extends Visitor {
    public MitarbeiterPrintVisitor() {
        this.register(Lohnempfaenger.class,
            new Consumer<Lohnempfaenger>() {
                @Override
                public void accept(Lohnempfaenger l) {
                    System.out.println(l.nr + " " + l.name + " " +
                        l.stdLohn + " " + l.anzStd);
                }
            });
        this.register(Gehaltsempfaenger.class,
            new Consumer<Gehaltsempfaenger>() {
                @Override
                public void accept(Gehaltsempfaenger g) {
                    System.out.println(g.nr + " " + g.name + " " +
                        g.gehalt);
                }
            });
    }
}
```

```
package appl;

import java.util.function.Consumer;

import util.Visitor;

public class MitarbeiterGehaltssummeVisitor extends Visitor {
    public MitarbeiterGehaltssummeVisitor() {
        this.register(Gehaltsempfaenger.class,
            new Consumer<Gehaltsempfaenger>() {
                @Override
                public void accept(Gehaltsempfaenger g) {
                    MitarbeiterGehaltssummeVisitor.this.summe +=
g.getVerdienst();
                }
            });
    }
    private int summe = 0;
    public int getSumme() {
        return this.summe;
    }
}
```

Die `iterate`-Methode der Klasse `Firma` wird schließlich wie folgt angepasst:

```
package appl;
// ...
public class Firma {
    // ...
    public void iterate(Visitor visitor) {
        for (final Mitarbeiter m : this.list)
            visitor.dispatch(m);
    }
}
```

4.3.5 Lambdas

Die konkreten Visitor-Klassen des letzten Abschnitts waren reichlich "verbose" – mit Lambdas lassen sich diese Klassen wesentlich schlanker formulieren:

```
package appl;
// ...
public class MitarbeiterPrintVisitor extends Visitor {
    public MitarbeiterPrintVisitor() {
        this.register(Lohnempfaenger.class,
            l -> System.out.println(l.nr + " " + l.name + " "
                + l.stdLohn + " " + l.anzStd));
        this.register(Gehaltsempfaenger.class,
            g -> System.out.println(g.nr + " " + g.name + " "
                + g.gehalt));
    }
}
```

```
package appl;
// ...
public class MitarbeiterGehaltssummeVisitor extends Visitor {
    public MitarbeiterGehaltssummeVisitor() {
        this.register(Gehaltsempfaenger.class,
            g -> this.summe += g.getVerdienst());
    }
    private int summe = 0;
    public int getSumme() {
        return this.summe;
    }
}
```

Wir haben hiermit eine typsichere Variante des Visitor-Patterns entwickelt, die gleichzeitig aber die Nachteile des Original-Patterns vermeidet (es gibt kein Interface, welches bei der Erweiterung der Klassenhierarchie der zu besuchenden Objekte ebenfalls erweitert werden müsste).

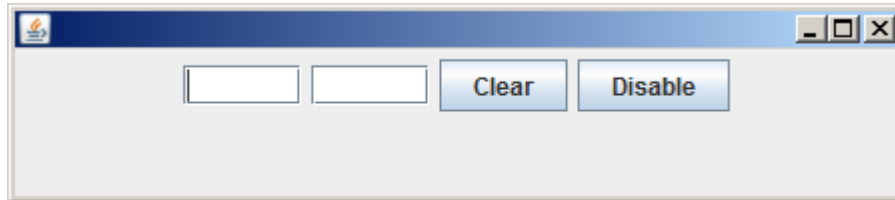
Frage: Was passiert bei folgendem MitarbeiterPrintVisitor?:

```
package appl;
// ...
public class MitarbeiterPrintVisitor extends Visitor {
    public MitarbeiterPrintVisitor() {
        this.register(Mitarbeiter.class,
            m -> System.out.println(m.nr + " " + m.name));
    }
}
```

4.3.6 Swing-Beispiel

Im folgenden wird eine kleine Utility-Klasse vorgestellt, die das Leben mit der GUI-Bibliothek Swing etwas vereinfacht.

Eine kleine Demo-Anwendung:



Diese Anwendung hat folgende Eigenschaften:

- wenn der Benutzer per Maus oder per Tab-Taste eines der beiden Eingabefelder betritt, wird dessen Inhalt selektiert (als blau markiert); wird das Feld wieder verlassen, so wird natürlich auch die Selektion verschwinden (der Hintergrund wird wieder weiß).
- die Betätigung des Clear-Buttons löscht alle Eingabefelder der Anwendung.
- die Betätigung des Disable-Buttons disabled alle Buttons...

Die Anwendung soll um weitere Eingabefelder und Buttons erweitert werden, ohne dass die Implementierung der o.g. Funktionalität immerzu angepasst werden muss.

Wir definieren eine von der `Visitor`-Klasse des letzten Abschnitts abgeleitete `AWTVisitor`-Klasse:

```
package util;

import java.awt.Component;
import java.awt.Container;

public class AWTVisitor extends Visitor {
    public void traverse(Component component) {
        this.dispatch(component);
        if (component instanceof Container) {
            final Container container = (Container) component;
            for (int i = 0; i < container.getComponentCount(); i++) {
                final Component child = container.getComponent(i);
                this.traverse(child);
            }
        }
    }
}
```

Die `traverse`-Methode traversiert den gesamten `Component`-Baum, dessen Wurzel der Methode übergeben wird. Jeden Knoten dieses Baumes wird an die von `Visitor` geerbte Methode `dispatch` übergeben. Man beachte die rekursive Implementierung der `traverse`-Methode.

Dann kann z.B. ein `Visitor` geschrieben werden, welcher die Inhalte alle Eingabefelder löscht, die in einem Komponentenbaum enthalten sind:

```
package appl;

import javax.swing.text.JTextComponent;
import util.AWTVisitor;

public class ClearTextFieldVisitor extends AWTVisitor {
    public ClearTextFieldVisitor() {
        this.register(JTextComponent.class, tf -> tf.setText(""));
    }
}
```

Oder einen, der alle Buttons eines Komponentenbaums enablen resp. disablen kann:

```
package appl;

import javax.swing.JButton;
import util.AWTVisitor;

public class EnableButtonVisitor extends AWTVisitor {
    private final boolean enable;

    public EnableButtonVisitor(boolean enable) {
        this.enable = enable;
        this.register(JButton.class, b -> b.setEnabled(this.enable));
    }
}
```

Oder einen `Visitor`, der bei allen Eingabefeldern eines Komponentenbaums einen "BlueWhiteHandler" registriert (der für das Blau- resp. das Weißmachen verantwortlich ist):

```
package appl;

import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;
import javax.swing.text.JTextComponent;
import util.AWTVisitor;

public class RegisterBlueWhiteHandlerVisitor extends AWTVisitor {

    private static FocusListener blueWhiteHandler = new
    FocusListener() {
        @Override
        public void focusGained(FocusEvent e) {
            ((JTextComponent) e.getSource()).selectAll();
        }
        @Override
        public void focusLost(FocusEvent e) {
            ((JTextComponent) e.getSource()).select(0, 0);
        }
    };

    public RegisterBlueWhiteHandlerVisitor() {
```

```
        this.register(JTextComponent.class,
            tf -> tf.addFocusListener(blueWhiteHandler));
    }
}
```

Hier der Quellcode der kleinen Demo-Anwendung:

```
package appl;

import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;

public class Application extends JFrame {

    public static void main(String[] args) {
        new Application();
    }

    private final JTextField textFieldNr = new JTextField(5);
    private final JTextField textFieldName = new JTextField(5);
    private final JButton buttonClear = new JButton("Clear");
    private final JButton buttonDisable = new JButton("Disable");

    public Application() {
        this.setLayout(new FlowLayout());
        this.add(this.textFieldNr);
        this.add(this.textFieldName);
        this.add(this.buttonClear);
        this.add(this.buttonDisable);

        new RegisterBlueWhiteHandlerVisitor().traverse(this);

        this.buttonClear.addActionListener(
            e -> new ClearTextFieldVisitor().traverse(this));

        this.buttonDisable.addActionListener(
            e -> new EnableButtonVisitor(false).traverse(this));

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setBounds(100, 100, 450, 100);
        this.setVisible(true);
    }
}
```

Man beachte, dass jederzeit neue Eingabefelder und Buttons hinzukommen können. Das in den Visitoren implementierte Verhalten bezieht sich immer auf alle Eingabefelder, auf alle Buttons etc...

4.4 Interpreter

"Definiere für eine gegebene Sprache eine Repräsentation der Grammatik sowie einen Interpreter, der die Repräsentation nutzt, um Sätze in der Sprache zu implementieren." (Gamma, 285)

4.4.1 Problem

Gegeben sei folgende DTD (`library.dtd`):

```
<!ELEMENT library (authors, books)>
<!ELEMENT books (book*)>
<!ELEMENT authors (author*)>
<!ELEMENT author (id, firstname, lastname)>
<!ELEMENT book (isbn, title, author-id, price)>
<!ELEMENT isbn (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author-id (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Die DTD legt die Syntax für die Beschreibung einer Bibliothek fest.

Das folgende XML-Dokument beschreibt eine konkrete Bibliothek (`library.xml`):

```
<?xml version='1.0'?>
<!DOCTYPE library SYSTEM "library.dtd">

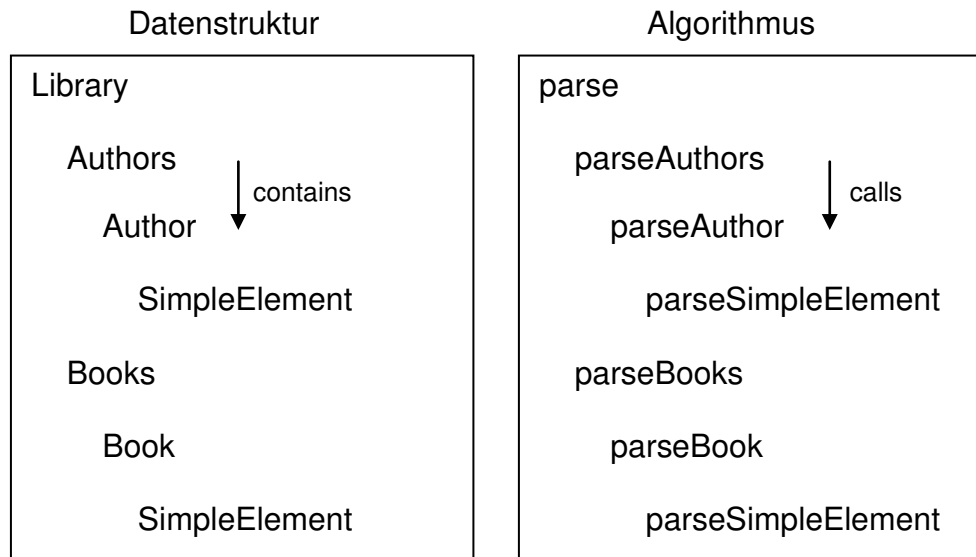
<library>
  <authors>
    <author>
      <id>1000</id>
      <firstname>Niklaus</firstname>
      <lastname>Wirth</lastname>
    </author>
    <author>
      <id>2000</id>
      <firstname>Bertrand</firstname>
      <lastname>Meyer</lastname>
    </author>
  </authors>
  <books>
    <book>
      <isbn>11111</isbn>
      <title>Modula</title>
      <author-id>1000</author-id>
      <price>2000</price>
    </book>
    <book>
      <isbn>22222</isbn>
      <title>Oberon</title>
      <author-id>1000</author-id>
      <price>3000</price>
    </book>
    <book>
      <isbn>33333</isbn>
      <title>Pascal</title>
      <author-id>1000</author-id>
      <price>4000</price>
    </book>
    <book>
      <isbn>44444</isbn>
```

```
        <title>Eiffel</title>
        <author-id>2000</author-id>
        <price>5000</price>
    </book>
</books>
</library>
```

Wie kann ein solcher Text mittels des `XMLScanners` geparkt werden?
Wie kann die Implementierung des Parsers an der Struktur der Eingabe ausgerichtet werden? Oder: wie kann die Programmstruktur von der Datenstruktur abgeleitet werden?

4.4.2 Lösung

Man schreibt einen `LibraryParser`, dessen Methoden-Struktur die Syntax des XML-Dokuments spiegelt:



```

package appl;

import java.io.InputStream;
import util.xml.XMLScanner;

public class LibraryParser {
    private final XMLScanner scanner;
    public LibraryParser(InputStream in) throws Exception {
        this.scanner = new XMLScanner(in);
    }
    public void parse() throws Exception {
        this.scanner.start("library");
        this.parseAuthors();
        this.parseBooks();
        this.scanner.end("library");
    }
    private void parseAuthors() throws Exception {
        this.scanner.start("authors");
        while (this.scanner.isStart("author"))
            this.parseAuthor();
        this.scanner.end("authors");
    }
    private void parseBooks() throws Exception {
        this.scanner.start("books");
        while (this.scanner.isStart("book"))
            this.parseBook();
        this.scanner.end("books");
    }
    private void parseAuthor() throws Exception {
        this.scanner.start("author");
        this.parseSimpleElement("id");
    }
}

```

```
        this.parseSimpleElement("firstname");
        this.parseSimpleElement("lastname");
        this.scanner.end("author");
    }
    private void parseBook() throws Exception {
        this.scanner.start("book");
        this.parseSimpleElement("isbn");
        this.parseSimpleElement("title");
        this.parseSimpleElement("author-id");
        this.parseSimpleElement("price");
        this.scanner.end("book");
    }
    private void parseSimpleElement(String name) throws Exception {
        final String text = this.scanner.startTextEnd(name);
        System.out.println("parseSimpleElement: " + name + " " + text);
    }
}
```

Man beachte den systematischen, von der zu parsenden Datenstruktur "abgeleiteten" Aufbau des Parsers.

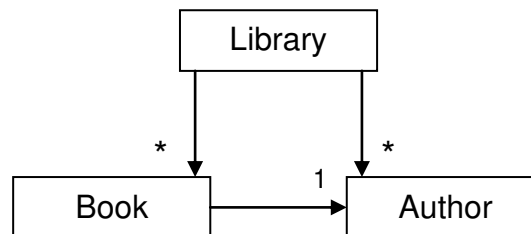
Hier die Anwendung (sie zeigt zunächst nur, dass die `parse`-Methode keine Exceptions wirft):

```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        final LibraryParser parser = new LibraryParser(
            new FileInputStream("src/library.xml"));
        parser.parse();
    }
}
```

4.4.3 Bau einer Datenstruktur

Der `LibraryParser` kann schließlich derart erweitert werden, dass er beim Ablauf des Parsens interne Datenstrukturen produziert:



Hier zunächst die Klassen `Author`, `Book` und `Library`:

```
package appl;

public class Author {

    private int id;
    private String firstname;
    private String lastname;

    // getter, setter, toString...
}
```

```
package appl;

public class Book {
    private String isbn;
    private String title;
    private double price;
    private Author author;

    // getter, setter, toString...
}
```

Man beachte, dass das Attribut `author` von Typ `Author` ist – es ist also kein einfacher `String`.

```
package appl;
//...
public class Library {

    private final List<Author> authors = new ArrayList<>();
    private final List<Book> books = new ArrayList<>();

    public void add(Author author) {
        this.authors.add(author);
    }
    public void add(Book book) {
        this.books.add(book);
    }

    public Author getAuthor(int id) {
```

```
        for (final Author author : this.authors)
            if (author.getId() == id)
                return author;
        return null;
    }

    public List<Author> getAuthors() {
        return Collections.unmodifiableList(this.authors);
    }
    public List<Book> getBooks() {
        return Collections.unmodifiableList(this.books);
    }
}
```

Hier nun der `LibraryParser`:

```
package appl;

import java.io.InputStream;

import util.xml.XMLScanner;

public class LibraryParser {

    private final XMLScanner scanner;
    private final Library library = new Library();

    public LibraryParser(InputStream in) throws Exception {
        this.scanner = new XMLScanner(in);
    }

    public Library parse() throws Exception {
        this.scanner.start("library");
        this.parseAuthors();
        this.parseBooks();
        this.scanner.end("library");
        return this.library;
    }

    private void parseAuthors() throws Exception {
        this.scanner.start("authors");
        while (this.scanner.isStart("author"))
            this.library.add(this.parseAuthor());
        this.scanner.end("authors");
    }

    private void parseBooks() throws Exception {
        this.scanner.start("books");
        while (this.scanner.isStart("book"))
            this.library.add(this.parseBook());
        this.scanner.end("books");
    }

    private Author parseAuthor() throws Exception {
        final Author author = new Author();
        this.scanner.start("author");
        author.setId(Integer.parseInt(this.parseSimpleElement("id")));
        author.setFirstname(this.parseSimpleElement("firstname"));
        author.setLastname(this.parseSimpleElement("lastname"));
        this.scanner.end("author");
        return author;
    }
}
```

```
private Book parseBook() throws Exception {
    final Book book = new Book();
    this.scanner.start("book");
    book.setIsbn(this.parseSimpleElement("isbn"));
    book.setTitle(this.parseSimpleElement("title"));
    book.setAuthor(this.library.getAuthor(
        Integer.parseInt(this.parseSimpleElement("author-id"))));
    book.setPrice(Double.parseDouble(
        this.parseSimpleElement("price")));
    this.scanner.end("book");
    return book;
}
private String parseSimpleElement(String name) throws Exception {
    return this.scanner.startTextEnd(name);
}
}
```

Man beachte die systematischen Erweiterungen – jede Parse-Methode erzeugt ein Resultat, welches in den übergeordneten parse-Methoden in ein jeweils komplexeres Resultat eingebunden wird.

Die Anwendung schließlich sieht wie folgt aus:

```
package appl;
// ...
public class Application {

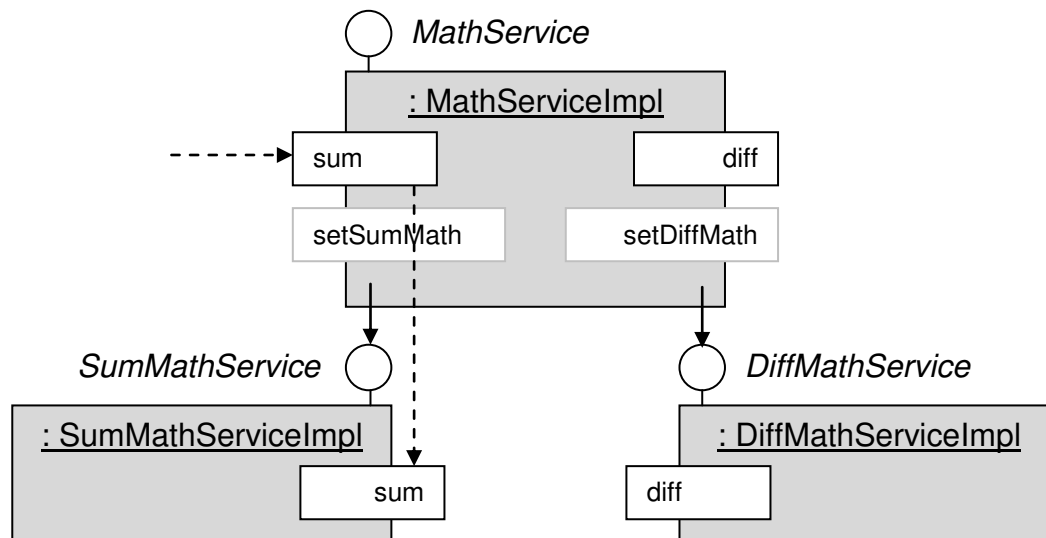
    public static void main(String[] args) throws Exception {
        final LibraryParser parser =
            new LibraryParser(new FileInputStream("src/library.xml"));
        final Library library = parser.parse();
        library.getAuthors().forEach(System.out::println);
        library.getBooks().forEach(System.out::println);
    }
}
```

Aufgaben

Entfernen Sie aus der XML-Datei (und natürlich auch aus der DTD) die `<authors>`- und `<books>`-Tags (sie sind ja eigentlich überflüssig). Und passen Sie dann den Parser an die geänderte Syntax an.

4.4.4 BeanFactory-Beispiel

Im folgenden wird ein elementarer Kern der sog. `BeanFactory` des Spring-Frameworks nachgebaut. Es geht hier u.a. ein wenig um das Thema "Dependency Injection".



Angenommen, es existieren folgende Interfaces:

```
package services;

public interface SumMathService {
    public abstract int sum(int x, int y);
}
```

```
package services;

public interface DiffMathService {
    public abstract int diff(int x, int y);
}
```

```
package services;

public interface MathService {
    public abstract int sum(int x, int y);
    public abstract int diff(int x, int y);
}
```


Diese Interfaces sind wie folgt implementiert:

```
package services.impl;  
  
import services.SumMathService;  
  
public class SumMathServiceImpl implements SumMathService{  
    @Override  
    public int sum(int x, int y) {  
        return x + y;  
    }  
}
```

```
package services.impl;  
  
import services.DiffMathService;  
  
public class DiffMathServiceImpl implements DiffMathService{  
    @Override  
    public int diff(int x, int y) {  
        return x - y;  
    }  
}
```

```
package services.impl;  
  
import services.DiffMathService;  
import services.MathService;  
import services.SumMathService;  
  
public class MathServiceImpl implements MathService{  
  
    private SumMathService sumMath;  
    private DiffMathService diffMath;  
  
    public void setSumMath(SumMathService sumMath) {  
        this.sumMath = sumMath;  
    }  
  
    public void setDiffMath(DiffMathService diffMath) {  
        this.diffMath = diffMath;  
    }  
  
    @Override  
    public int sum(int x, int y) {  
        return this.sumMath.sum(x, y);  
    }  
  
    @Override  
    public int diff(int x, int y) {  
        return this.diffMath.diff(x, y);  
    }  
}
```

Ein `MathServiceImpl`-Objekt ist offenbar abhängig von zwei weiteren Objekten – von einem Objekt, dessen Klasse das Interface `SumMathService` implementiert, und von einem zweiten Objekt, dessen Klasse `DiffMathService` implementiert.

Welche konkreten Klassen diese Interfaces implementieren (welche konkreten also vom `MathServiceImpl` genutzt werden), soll dem "Chefmathematiker" nicht bekannt sein. Deshalb besitzt er zwei Methoden, mittels derer seine `sumMath` und seine `diffMath`-Variable initialisiert werden können: `setSumMath` und `setDiffMath`.

Natürlich könnte man nun z.B. am Anfang der `main`-Methode die drei `Impl`-Objekte erzeugen und dann sie `set`-Methoden auf den `MathServiceImpl` aufrufen, um dem Chef die beiden Hilfsarbeiter zu "injizieren". Man kann diese "Konfiguration" aber auch in einer XML-Datei spezifizieren:

```
<?xml version='1.0'?>
<!DOCTYPE library SYSTEM "beans.dtd">

<beans>

    <bean id = "sumMathService"
class="services.impl.SumMathServiceImpl"/>

    <bean id = "diffMathService"
class="services.impl.DiffMathServiceImpl"/>

    <bean id = "mathService" class="services.impl.MathServiceImpl">
        <property name="sumMath" ref="sumMathService"/>
        <property name="diffMath" ref="diffMathService"/>
    </bean>

</beans>
```

Aufgrund dieser XML-Datei kann eine allgemeine "Bean-Factory" die drei `Impl`-Objekte erzeugen und unter bestimmten IDs registrieren. Auf den Chef-Mathematiker können dann via Reflection die Properties `sumMath` und `diffMath` gesetzt werden (also die Methoden `setSumMath` und `setDiffMath` aufgerufen werden). Beim Aufruf dieser Methoden werden dann als Parameter diejenigen Objekte übergeben, die unter den IDs `sumMathService` und `diffMathService` registriert wurden.

Der Parser wird eine Datenstruktur erzeugen, welche durch folgenden Klassen definiert ist:

```
package util;

public class Property {

    private final String name;
    private final String ref;

    public Property(String name, String ref) {`... }
```

```

public String getName() { return this.name; }
public String getRef() { return this.ref; }

@Override
public String toString() { ... }
}

```

```

package util;
//
public class Bean {

    private final String id;
    private final String className;
    private final List<Property> properties;

    public Bean(String id, String className,
                List<Property> properties) {
        this.id = id;
        this.className = className;
        this.properties = new ArrayList<>(properties);
    }

    public String getId() { return this.id; }
    public String getClassName() { return this.className; }

    public List<Property> getProperties() {
        return Collections.unmodifiableList(this.properties);
    }

    @Override
    public String toString() { ... }
}

```

```

package util;
// ...
public class Beans {

    private final List<Bean> beans;

    public Beans(List<Bean> beans) {
        this.beans = new ArrayList<>(beans);
    }

    public List<Bean> getBeans() {
        return Collections.unmodifiableList(this.beans);
    }

    @Override
    public String toString() { ... }
}

```

Man beachte, dass die Objekte dieser Klassen allesamt immutable sind.

Man beachte weiterhin, dass die Datenstruktur eins-zu-eins von der Struktur der XML-Datei "abgeleitet" ist.

Hier der `BeansParser`, dessen `parse`-Methode ein `Beans`-Objekt zurückliefert (also ein Objekt, welches die gesamten Daten einer XML-Datei repräsentiert):

```
package util;
// ...
public class BeansParser {

    final XMLScanner scanner;

    public BeansParser(InputStream in) {
        this.scanner = new XMLScanner(in);
    }

    public Beans parse() {
        final List<Bean> beans = new ArrayList<>();
        this.scanner.start("beans");
        while (this.scanner.isStart("bean"))
            beans.add(this.parseBean());
        this.scanner.end("beans");
        return new Beans(bbeans);
    }

    private Bean parseBean() {
        final List<Property> properties = new ArrayList<>();
        final Map<String,String> attributes =
            this.scanner.getAttributes();
        this.scanner.start("bean");
        while (this.scanner.isStart("property"))
            properties.add(this.parseProperty());
        this.scanner.end("bean");
        return new Bean(
            attributes.get("id"),
            attributes.get("class"),
            properties);
    }

    private Property parseProperty() {
        final Map<String,String> attributes =
            this.scanner.getAttributes();
        this.scanner.start("property");
        this.scanner.end("property");
        return new Property(
            attributes.get("name"),
            attributes.get("ref"));
    }
}
```

Auch der Parser ist eins-zu-eins von der Datenstruktur "abgeleitet".

Hier schließlich die eigentliche `BeanFactory`:

```
package util;
// ...
public class BeanFactory {

    private final Map<String, Object> objects = new HashMap<>();

    public BeanFactory(InputStream in) throws Exception {
```

```
final BeansParser parser = new BeansParser(in);
final Beans beans = parser.parse();
for (final Bean bean : beans.getBeans()) {
    final Object object =
        Class.forName(bean.getClassName()).newInstance();
    this.objects.put(bean.getId(), object);
}
for (final Bean bean : beans.getBeans()) {
    final Class<?> beanClass =
        Class.forName(bean.getClassName());
    for (final Property property : bean.getProperties()) {
        final Object obj =
            this.objects.get(property.getRef());
        if (obj == null)
            throw new RuntimeException("cannot resolve ref='" +
                property.getRef() + "' for bean '" +
                bean.getId() + "'");
        final Method method = findSetter(
            beanClass, property.getName(), obj.getClass());
        if (method == null)
            throw new RuntimeException("cannot initialize '" +
                property.getName() + "' of bean '" +
                bean.getId() + "'");
        method.invoke(this.objects.get(bean.getId()), obj);
    }
}

public Object getBean(String id) {
    return this.objects.get(id);
}

private static Method findSetter(Class<?> cls,
    String propertyName, Class<?> paramClass) {
    final String setterName = "set" +
        Character.toUpperCase(propertyName.charAt(0)) +
        propertyName.substring(1);
    for (final Method method : cls.getMethods()) {
        if (!method.getName().equals(setterName))
            continue;
        if (method.getParameterTypes().length != 1)
            continue;
        if (!method.getParameterTypes()[0].
            isAssignableFrom(paramClass))
            continue;
        return method;
    }
    return null;
}
}
```

Die `BeanFactory` parst die XML-Datei und verwendet das dabei erzeugt Resultat (das `Beans`-Objekt) zum Aufbau einer eigenen Datenstruktur (der `objects`-Registry). Für jeden `Bean`-Eintrag wird ein entsprechendes Objekt erzeugt. Falls in der XML spezifiziert, werden auf die erzeugten Objekte die Property-Methoden aufgerufen und auf diese Weise die Objekte miteinander verbunden.

Eine einfache Anwendung:

```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {

        final BeanFactory beanFactory =
            new BeanFactory(new FileInputStream("src/beans.xml"));

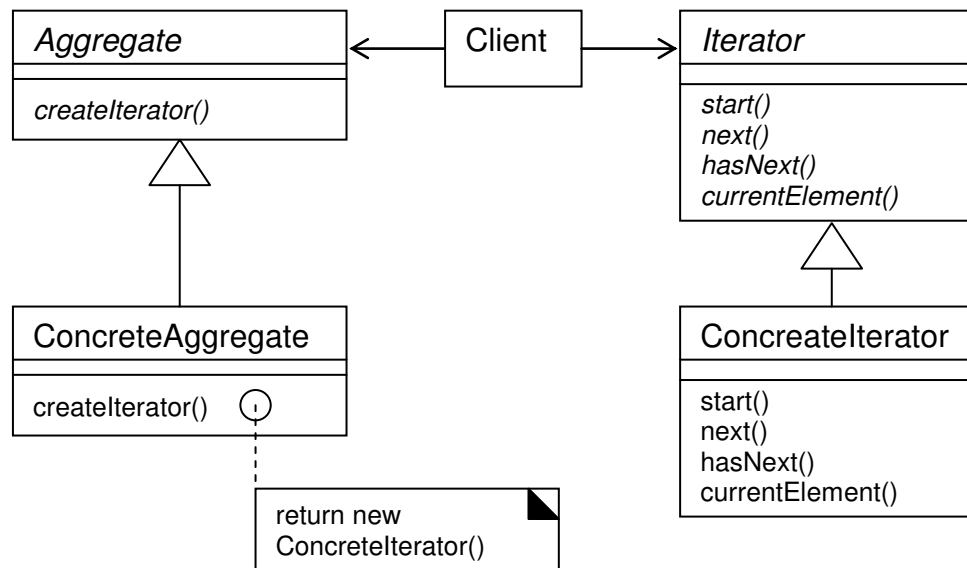
        final MathService mathService =
            (MathService) beanFactory.getBean("mathService");

        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

Die Anwendung benutzt ausschließlich das Interface `MathService`. Sie kennt keinerlei Implementierungsklassen. Und genau das ist die Motivation für eine `BeanFactory`.

4.5 Iterator

"Ermögliche den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen." (Gamma, 299)



4.5.1 Problem

Die folgende Klasse `SimpleArrayList` ist ein "Schmalspur-Version" der Standardklasse `ArrayList`:

```
package util;

public class SimpleArrayList<T> {

    private T[] elements = this.createArray(2);
    private int size;

    public void add(T element) {
        this.ensureCapacity();
        this.elements[this.size] = element;
        this.size++;
    }

    public int size() {
        return this.size;
    }

    public T get(int index) {
        if (index < 0 || index >= this.size)
            throw new IndexOutOfBoundsException();
        return this.elements[index];
    }

    private void ensureCapacity() {
        if (this.size == this.elements.length) {
            final T[] newElements = this.createArray(2 * this.size);
            for (int i = 0; i < this.size; i++)
                newElements[i] = this.elements[i];
            this.elements = newElements;
        }
    }

    @SuppressWarnings("unchecked")
    private T[] createArray(int length) {
        return (T[]) new Object[length];
    }
}
```

Eine `SimpleArrayList<T>` kann Elemente des Typs `T` (oder Elemente eines Typs, der zu `T` kompatibel ist) enthalten. Die Elemente werden in einem gewöhnlichen Java-Array gehalten, der bei Bedarf durch einen größeren Array ersetzt wird (wobei die Elemente des alten Arrays in den neuen Array kopiert werden) – siehe die private Methode `ensureCapacity`.

Per `add` kann ein Element ans Ende der Datenstruktur angefügt werden; `size` liefert die Anzahl der Elemente und die `get`-Methode das `index`-te Element zurück.

Wir implementieren auch ein Schmalspur-Version der Standardklasse `LinkedList`:

```
package util;

public class SimpleLinkedList<T> {

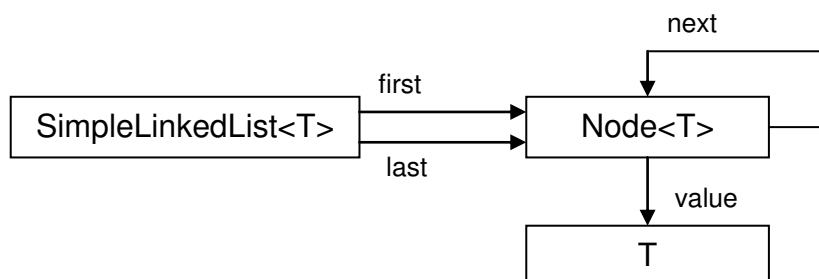
    public static class Node<T> {
        private final T value;
        private Node<T> next;
        public Node(T value) {
            this.value = value;
        }
        public T getValue() {
            return this.value;
        }
        public Node<T> getNext() {
            return this.next;
        }
    }

    private Node<T> first;
    private Node<T> last;

    public void add(T element) {
        final Node<T> node = new Node<T>(element);
        if (this.first == null)
            this.first = node;
        else
            this.last.next = node;
        this.last = node;
    }

    public Node<T> getFirst() {
        return this.first;
    }
}
```

Hier ein Klassendiagramm für die `SimpleLinkedList`:



Eine Anwendung möchte nun sowohl eine `SimpleArrayList` als auch eine `SimpleLinkedList` verwenden. Nachdem jeweils einige Elemente zu den Listen hinzugefügt wurden, sollen die Elemente beider Listen sequentiell ausgegeben werden:

```
package appl;

import util.SimpleArrayList;
import util.SimpleLinkedList;

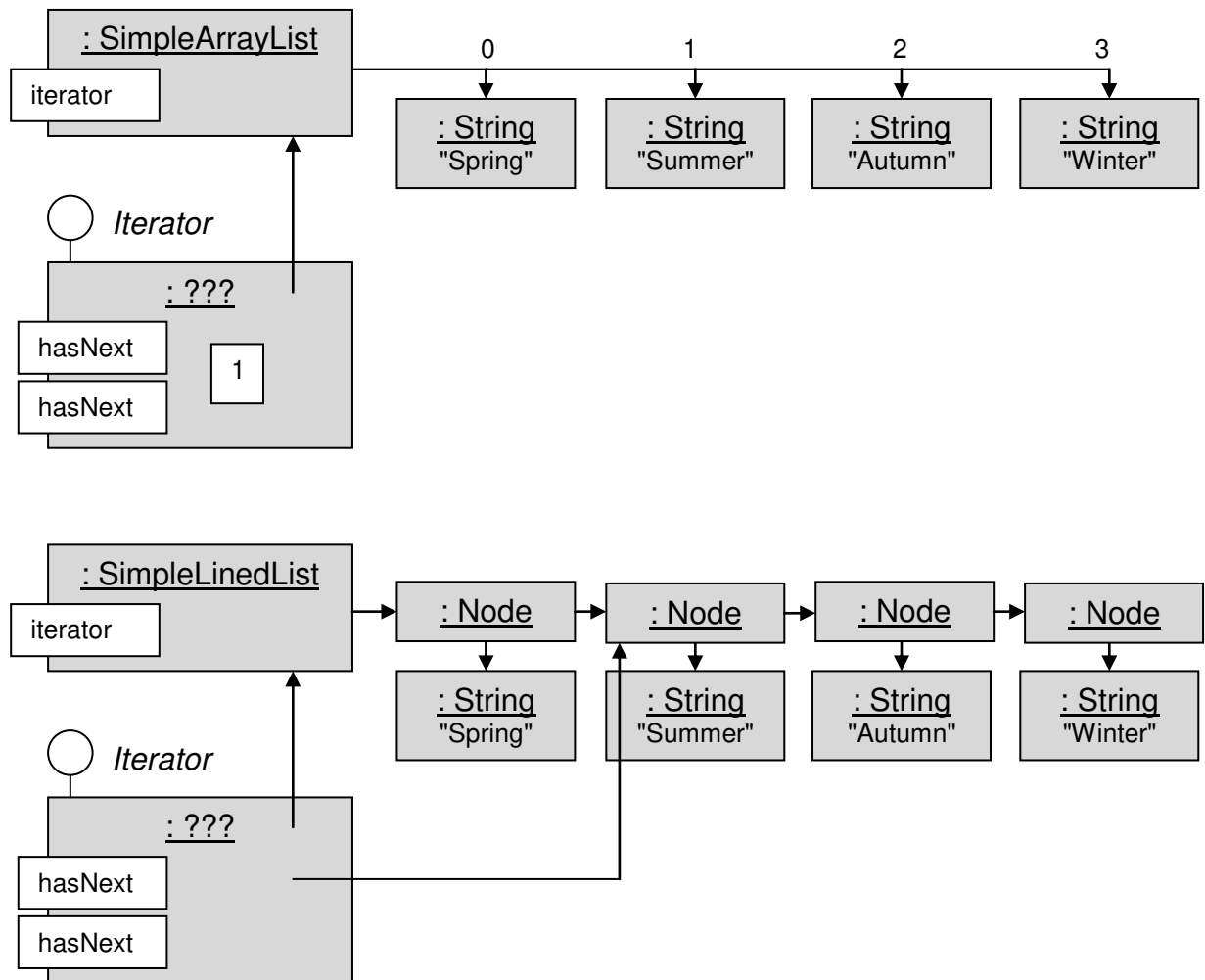
public class Application {

    public static void main(String[] args) throws Exception {
        final SimpleArrayList<String> l1 = new SimpleArrayList<>();
        l1.add("spring");
        l1.add("summer");
        l1.add("autumn");
        l1.add("winter");
        for (int i = 0; i < l1.size(); i++) {
            final String s = l1.get(i);
            System.out.println(s);
        }
        final SimpleLinkedList<String> l2 = new SimpleLinkedList<>();
        l2.add("spring");
        l2.add("summer");
        l2.add("autumn");
        l2.add("winter");
        for(SimpleLinkedList.Node<String> node = l2.getFirst();
            node != null; node = node.getNext()) {
            final String s = node.getValue();
            System.out.println(s);
        }
    }
}
```

Das Hinzufügen sieht bei beiden Listen gleich aus (Aufruf der `add`-Methode). Leider benötigt man zwei unterschiedliche Verfahren, um die Inhalte dieser Listen sequentiell auszugeben. Es wäre natürlich wünschenswert, wenn beide Listen ein gleiches Verfahren ermöglichten.

4.5.2 Lösung

Die beiden List-Klassen werden ergänzt durch eine Methode, welche einen `Iterator` liefert. `Iterator` ist bekanntlich ein Interface der Standardbibliothek. Ein `Iterator`-Objekt ermöglicht den sequentiellen Zugriff auf die Elemente einer Collection.



```
package util;

import java.util.Iterator;

public class SimpleArrayList<T> {

    private T[] elements = this.createArray(2);
    private int size;

    // wie gehabt...

    public Iterator<T> iterator() {
        return new Iterator<T>() {
            int currentIndex = 0;

            @Override
            public boolean hasNext() {
                return this.currentIndex < SimpleArrayList.this.size;
            }

            @Override
            public T next() {
                final T element =
                    SimpleArrayList.this.elements[this.currentIndex];
                this.currentIndex++;
                return element;
            }
        };
    }
}
```

`iterator` liefert ein Objekt einer von `Iterator<T>`-abgeleiteten anonymen Klasse zurück.. Ein solches Objekt dient als "Lesezeichen". Es besitzt einen `currentIndex`, welcher das zuletzt gelesene Element markiert. Die `hasNext`-Methode prüft, ob bereits alle Elemente abgearbeitet wurden. Die `next`-Methode liefert das Objekt zurück, welches sich an der aktuellen Position befindet und schiebt das Lesezeichen eine Position weiter.

Auch die Klasse `SimpleLinkedList` kann um eine Methode erweitert werden, welche einen `Iterator` liefert (sinnvollerweise wird sie auch hier als `iterator` bezeichnet):

```
package util;

import java.util.Iterator;

public class SimpleLinkedList<T> {

    public static class Node<T> {
        private final T value;
        private Node<T> next;

        // wie gehabt
    }

    private Node<T> first;
    private Node<T> last;

    // wie gehabt

    public Iterator<T> iterator() {
        return new Iterator<T>() {

            Node<T> currentNode = SimpleLinkedList.this.first;

            @Override
            public boolean hasNext() {
                return this.currentNode != null;
            }

            @Override
            public T next() {
                final T element = this.currentNode.value;
                this.currentNode = this.currentNode.next;
                return element;
            }

        };
    }
}
```

Der `Iterator` einer `SimpleLinkedList` ist natürlich vollständig anders implementiert als derjenige der Klasse `SimpleArrayList` – der Zustand der `Iterators` wird hier nicht in einem Index gehalten, sondern in einer Referenz auf den aktuellen `Node`. Beide `Iteratoren` sehen von "außen" aber gleich aus – beide implementieren das Interface `Iterator`.

Nun können wir in der gleichen Weise auf die Elemente beider Listen sequentiell zugreifen:

```
package appl;

import java.util.Iterator;
import util.SimpleArrayList;
import util.SimpleLinkedList;

public class Application {

    public static void main(String[] args) throws Exception {
        final SimpleArrayList<String> l1 = new SimpleArrayList<>();
        l1.add("spring");
        l1.add("summer");
        l1.add("autumn");
        l1.add("winter");
        print(l1.iterator());

        final SimpleLinkedList<String> l2 = new SimpleLinkedList<>();
        l2.add("spring");
        l2.add("summer");
        l2.add("autumn");
        l2.add("winter");
        print(l2.iterator());
    }

    private static <T> void print(Iterator<T> iter) {
        while(iter.hasNext()) {
            final T value = iter.next();
            System.out.println(value);
        }
    }
}
```

Der `print`-Methode wird nun einfach ein `Iterator<T>` übergeben – der Methode kann es dann egal sein, ob dieser Iterator von `SimpleArrayList<T>` oder von `SimpleLinkedList<T>` stammt. Mittels der Kombination `hasNext` und `next` können wir sequentiell auf die Elemente der jeweiligen List zugreifen – ohne den tatsächlichen Typ der Liste kennen zu müssen. (Natürlich können auch die Standardklassen `ArrayList` und `LinkedList` einen `Iterator` liefern...)

4.5.3 Das Interface Iterable

In Java 5 wurde das neue Interface `java.lang.Iterable` eingeführt. Eine Klasse, welche dieses Interface implementiert, muss eine `iterator`-Methode implementieren, welche einen `Iterator` zurückliefert.

Die beiden List-Klassen des letzten Abschnitts waren also bereits "Inoffiziell" `Iterable` – aber noch nicht "offiziell".

Wir erweitern also die Überschriften der beiden Klassen:

```
public class SimpleArrayList<T> implements Iterable<T> {
    // ...
}
```

```
public class SimpleLinkedList<T> implements Iterable<T> {
    // ...
}
```

Dann können wir die Application umschreiben:

```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        final SimpleArrayList<String> l1 = new SimpleArrayList<>();
        l1.add("spring");
        l1.add("summer");
        l1.add("autumn");
        l1.add("winter");
        printA(l1);
        printB(l1);

        final SimpleLinkedList<String> l2 = new SimpleLinkedList<>();
        l2.add("spring");
        l2.add("summer");
        l2.add("autumn");
        l2.add("winter");
        printA(l2);
        printB(l1);
    }

    private static <T> void printA(Iterable<T> iterable) {
        final Iterator<T> iter = iterable.iterator();
        while(iter.hasNext()) {
            final T value = iter.next();
            System.out.println(value);
        }
    }

    private static <T> void printB(Iterable<T> iterable) {
        for(final T value : iterable)
            System.out.println(value);
    }
}
```

`Application` definiert nun zwei `Print`-Methoden: `printA` und `printB`. Beiden Methoden wird nun statt eines `Iterators` ein `Iterable` übergeben. Beim Aufruf kann also direkt jeweils eine der beiden Listen übergeben werden.

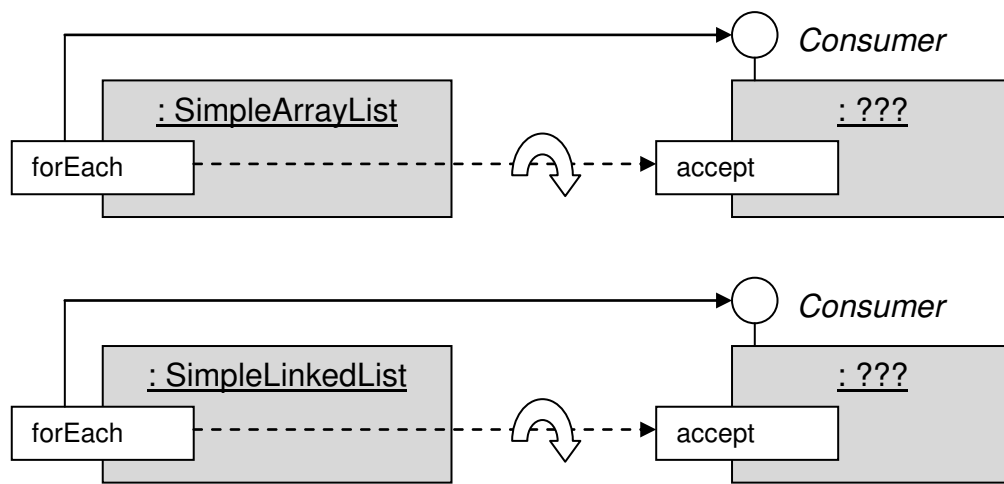
`printA` ruft auf das übergebene `Iterable` explizit die `iterator`-Methode auf und benutzt explizit die Methoden des von ihr gelieferten `IteratorS`.

`printB` benutzt die in Java 5 eingeführte "for-each"-Schleife. Um diese Schleife verwenden zu können, muss das hinter dem Doppelpunkt angegebene Objekt das Interface `Iterable` implementieren. Die Schleife wird dann auf die `Iterator`-basierte Schleife zurückgeführt. Intern passiert also bei `printB` dasselbe wie bei `printA`.

4.5.4 Interner Iterator

Die Implementierung eines `Iterators` ist nicht immer einfach. Bei indizierbaren oder einfach verketteten Datenstrukturen ist die Sache relativ problemlos – bei Bäumen aber z.B. recht aufwendig.

Statt eines expliziten `Iterators` (auch als "externer" Iterator bezeichnet) kann dann ein "interner" Iterator verwendet werden. Auch dieser Iterator sei an den beiden Demo-Klassen `SimpleArrayList` und `SimpleLinkedList` demonstriert.



Beide Klassen werden um ein `forEach`-Methode übergeben, welcher ein `Consumer` übergeben wird (`Consumer` ist ein in Java 8 neu eingeführtes Interface, welches eine einzige Methode enthält: `accept`):

```
package util;
// ...
import java.util.function.Consumer;

public class SimpleArrayList<T> {

    // wie gehabt...

    public void forEach(Consumer<T> consumer) {
        for(int i = 0; i < this.size; i++)
            consumer.accept(this.elements[i]);
    }
}
```

```
package util;
// ...
import java.util.function.Consumer;

public class SimpleLinkedList<T> {

    // wie gehabt...

    public void forEach(Consumer<T> consumer) {
```

```
        for(Node<T> node = this.first; node != null; node = node.next)
            consumer.accept(node.value);
    }
}
```

Die `forEach`-Methode implementiert jeweils eine Schleife, welche über alle Elemente der Liste iteriert und jedes Element an die `accept`-Methode des an `forEach` übergebenen `Consumer`s übergibt.

(Der `consumer`-Parameter sollte vielleicht besser als `Consumer<? super T>` definiert werden...)

Eine Anwendung:

```
package appl;

import java.util.function.Consumer;
// ...
public class Application {

    public static void main(String[] args) {
        final SimpleArrayList<String> l1 = new SimpleArrayList<>();
        l1.add("spring");
        l1.add("summer");
        l1.add("autumn");
        l1.add("winter");
        l1.forEach(new Consumer<String>() {
            @Override
            public void accept(String value) {
                System.out.println(value);
            }
        });
        l1.forEach(value -> System.out.println(value));
        l1.forEach(System.out::println);

        final SimpleLinkedList<String> l2 = new SimpleLinkedList<>();
        l2.add("spring");
        l2.add("summer");
        l2.add("autumn");
        l2.add("winter");
        l2.forEach(new Consumer<String>() {
            @Override
            public void accept(String value) {
                System.out.println(value);
            }
        });
        l2.forEach(value -> System.out.println(value));
        l2.forEach(System.out::println);
    }
}
```

Man kann an `foreach` ein Objekt einer anonymen Klasse übergeben, welche das Interface `Consumer` implementiert – z.B.:

```
l2.forEach(new Consumer<String>() {  
    @Override  
    public void accept(String value) {  
        System.out.println(value);  
    }  
});
```

Oder man kann einfach einen Lambda-Ausdruck übergeben (was natürlich viel eleganter ist) – z.B.:

```
l2.forEach(value -> System.out.println(value));
```

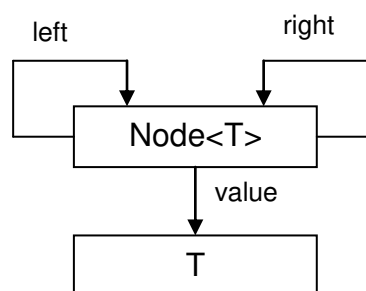
Oder man kann – noch einfacher – eine Methoden-Referenz übergeben:

```
l2.forEach(System.out::println);
```

4.5.5 Tree-Iterator

Im folgenden wird gezeigt, wie ein interner Iterator bei einem Binärbaum implementiert werden kann (die Implementierung eines externen Iterators ist bei einem solchen Baum nicht ganz einfach).

Ein Baum ist ein Knoten, welcher einen Wert enthält und zwei Referenzen besitzt (die möglicherweise null sind): eine auf einen linken (Teil-)Baum und eine auf einen rechten (Teil-)Baum:



```
package util;

import java.util.function.Consumer;

public class Node<T> extends Comparable<T> {
    private Node<T> left;
    private Node<T> right;
    private final T value;

    public Node(T value) { this.value = value; }

    public Node<T> getLeft() { return this.left; }
    public Node<T> getRight() { return this.right; }

    public void add(T value) {
        if (this.value.compareTo(value) > 0) {
            if (this.left == null)
                this.left = new Node<T>(value);
            else
                this.left.add(value);
        }
        else {
            if (this.right == null)
                this.right = new Node<T>(value);
            else
                this.right.add(value);
        }
    }

    public void foreach(Consumer<T> consumer) {
        if (this.left != null)
            this.left.foreach(consumer);
        consumer.accept(this.value);
        if (this.right != null)
            this.right.foreach(consumer);
    }
}
```

Hier eine Anwendung:

```
package appl;

import util.Node;

public class Application {

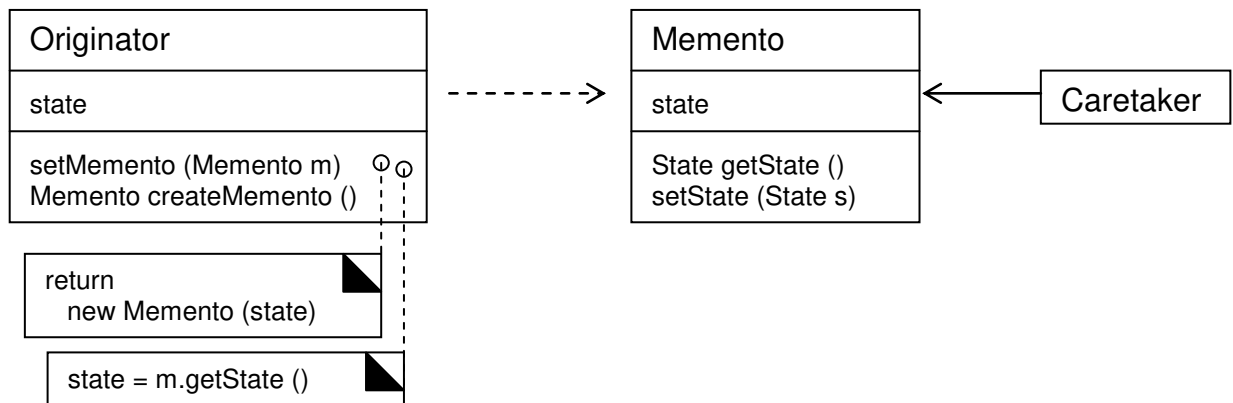
    public static void main(String[] args) throws Exception {
        final Node<Integer> root = new Node<Integer>(50);
        root.add(30);
        root.add(20);
        root.add(40);
        root.add(10);
        root.add(80);
        root.add(70);
        root.add(90);
        root.add(60);
        root.foreach(System.out::println);
    }
}
```

Die Ausgaben:

```
10
20
30
40
50
60
70
80
90
```

4.6 Memento

"Erfasse und externalisiere den internen Zustand eines Objekts, ohne seine Kapselung zu verletzen, so dass ein Objekt später in diesen Zustand zurückversetzt werden kann." (Gamma, 317)



4.6.1 Problem

Textdokumente seien durch folgende Klasse beschrieben:

```
package util;
// ...
public class Document {

    public static interface Element {
        void display();
    }

    public static class TextElement implements Element {
        private final StringBuilder builder = new StringBuilder();

        public TextElement(String s) {
            this.builder.append(s);
        }
        public void append(String s) {
            this.builder.append(s);
        }
        public void remove(int startIndex, int length) {
            this.builder.delete(startIndex, startIndex + length);
        }
        @Override
        public void display() {
            System.out.println(this.builder);
        }
    }

    public static class FontElement implements Element {
        private String fontname;
        private int fontsize;

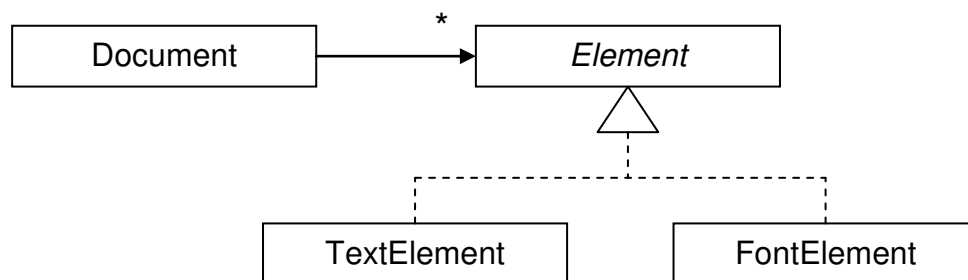
        public FontElement(String fontname, int fontsize) {
            this.fontname = fontname;
            this.fontsize = fontsize;
        }

        public String getFontname() {
            return this.fontname;
        }
        public void setFontname(String fontname) {
            this.fontname = fontname;
        }
        public int getFontsize() {
            return this.fontsize;
        }
        public void setFontsize(int fontsize) {
            this.fontsize = fontsize;
        }
        @Override
        public void display() {
            System.out.println(
                "[" + this.fontname + " " + this.fontsize + "]");
        }
    }

    private List<Element> elements = new ArrayList<>();
```

```
public void append(Element element) {  
    this.elements.add(element);  
}  
public void remove(int index) {  
    this.elements.remove(index);  
}  
public int size() {  
    return this.elements.size();  
}  
public Element get(int index) {  
    return this.elements.get(index);  
}  
public void display() {  
    for (final Element element : this.elements)  
        element.display();  
}  
}
```

Ein `Document` besteht aus einer Folge `Element` kompatibler Objekte – aus `TextElement`- und `FontElement`-Objekten:



Eine kleine Test-Anwendung:

```
package appl;  
  
import util.Document;  
  
public class Application {  
    public static void main(String[] args) throws Exception {  
        final Document doc = new Document();  
        doc.append(new Document.FontElement("Arial", 12));  
        doc.append(new Document.TextElement("Hello"));  
        doc.append(new Document.FontElement("Courier", 14));  
        doc.append(new Document.TextElement("World"));  
        doc.remove(2);  
        doc.display();  
    }  
}
```

Die Ausgaben:

```
[Arial 12]  
Hello  
World
```


Es wäre wünschenswert, wenn man einen Zwischenzustand eines Dokuments sichern könnte – um dann spätere Änderungen verwerfen zu können und wieder neu auf diesen Zwischenzustand aufzusetzen. (Natürlich kann man dann auch viele Zwischenzustände speichern wollen.)

(By the way: Wie könnte man erreichen, dass Dokumente geschachtelt werden können? Beachten Sie das Interface `Element`! Und denken Sie an das Composite-Pattern!)

4.6.2 Lösung

Die Klasse `Document` wird um eine innere Klasse `Memento` erweitert:

```
package util;
// ...
public class Document {

    public static interface Element { ... }

    public static class TextElement implements Element { ... }

    public static class FontElement implements Element { ... }

    public static class Memento {

        private final List<Element> elements;

        public Memento(Document doc) {
            this.elements = new ArrayList<Element>(doc.elements);
        }

        public void restore(Document doc) {
            doc.elements = this.elements;
        }

    }

    private List<Element> elements = new ArrayList<>();

    // wie gehabt...
}
```

Da `Memento` eine innere Klasse von `Document` ist, hat sie auch Zugriff auf die privaten Elemente der umschließenden Klasse – auf das `elements`-Feld. Ihr Konstruktor erstellt eine tiefe Kopie der `elements` des ihm übergebenen `Documents` und speichert diese Kopie in einer eigenen Instanzvariablen. Die `Restore`-Methode trägt in das ihr übergebene `Document` wieder diesen Zustand ein – ersetzt also den Zustand des ihr übergebenen `Documents` durch den vom `Memento` gespeicherten Zustand (wieder per tiefer Kopie).

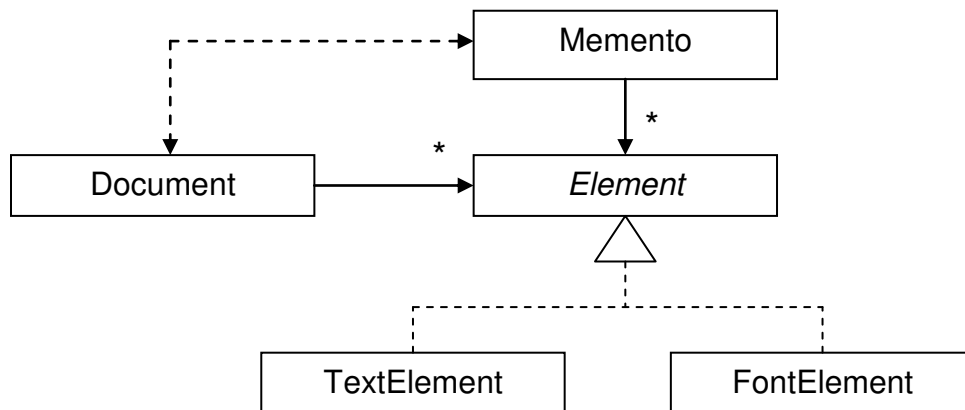
Ein `Caretaker` kümmert sich um das `Memento` (oder die vielen `Mementos`):

```
package util;

public class Caretaker {
    private Document.Memento memento;
    public Document.Memento getMemento() {
        return this.memento;
    }
    public void setMemento(Document.Memento memento) {
        this.memento = memento;
    }
}
```

(Der `Caretaker` könnte auch intelligenter sein – siehe den folgenden Abschnitt.)

Das Klassendiagramm:



Hier die Test-Anwendung:

```

package appl;

import util.Caretaker;
import util.Document;

public class Application {

    public static void main(String[] args) throws Exception {
        final Document doc = new Document();
        doc.append(new Document.FontElement("Arial", 12));
        doc.append(new Document.TextElement("Hello"));
        doc.append(new Document.FontElement("Courier", 14));
        doc.append(new Document.TextElement("World"));
        final Caretaker caretaker1 = new Caretaker();
        caretaker1.setMemento(new Document.Memento(doc));
        doc.append(new Document.TextElement("Spring"));
        doc.append(new Document.TextElement("Summer"));
        doc.display();
        System.out.println("-----");
        caretaker1.getMemento().restore(doc);
        doc.display();
    }
}
  
```

Die Ausgaben:

[Arial 12]

Hello

[Courier 14]

World

Spring

Summer

[Arial 12]

Hello

[Courier 14]

World

4.6.3 Serialisierung

Hier ein etwas intelligenterer `Caretaker`, der den Zustand eine `Documents` serialisiert auf Platte schreibt:

```
package util;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Caretaker {

    private final String filename;

    public Caretaker(String filename) {
        this.filename = filename;
    }

    public Document.Memento getMemento() throws Exception {
        try(ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream(this.filename))) {
            return (Document.Memento )ois.readObject();
        }
    }

    public void setMemento(Document.Memento memento) throws Exception {
        try(ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(this.filename))) {
            oos.writeObject(memento);
        }
    }
}
```

Der `Caretaker` selbst braucht nur mehr den Namen der Datei zu speichern, in welcher der `Document`-Zustand serialisiert ist.

Allerdings müssen nun einige Typen das Interface `Serializable` implementieren: `Document`, `Document.Element` und `Document.Memento`.

Und hier die geänderte Test-Anwendung:

```
package appl;

import util.Caretaker;
import util.Document;

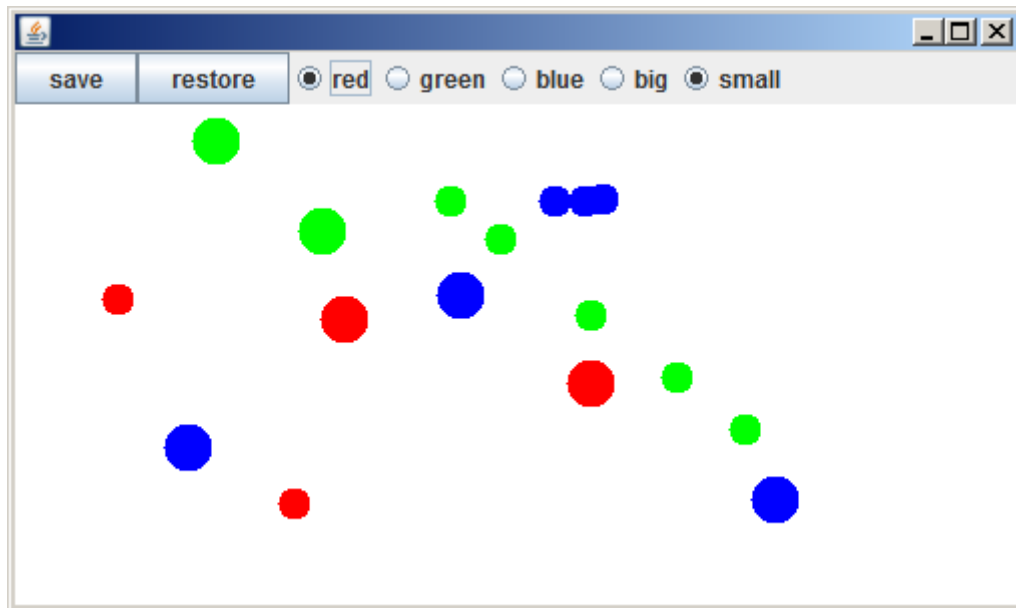
public class Application {

    public static void main(String[] args) throws Exception {
        final Document doc = new Document();
        doc.append(new Document.FontElement("Arial", 12));
        doc.append(new Document.TextElement("Hello"));
        doc.append(new Document.FontElement("Courier", 14));
        doc.append(new Document.TextElement("World"));
        final Caretaker caretaker1 = new Caretaker("src/doc.dat");
        caretaker1.setMemento(new Document.Memento(doc));
        doc.append(new Document.TextElement("Spring"));
        doc.append(new Document.TextElement("Summer"));
        doc.display();
        System.out.println("-----");
        caretaker1.getMemento().restore(doc);
        doc.display();
    }
}
```

(Wie könnte eine `Caretaker` sich um mehrere Zwischenzustände eines `Documents` kümmern?)

4.6.4 Ein grafischer Editor

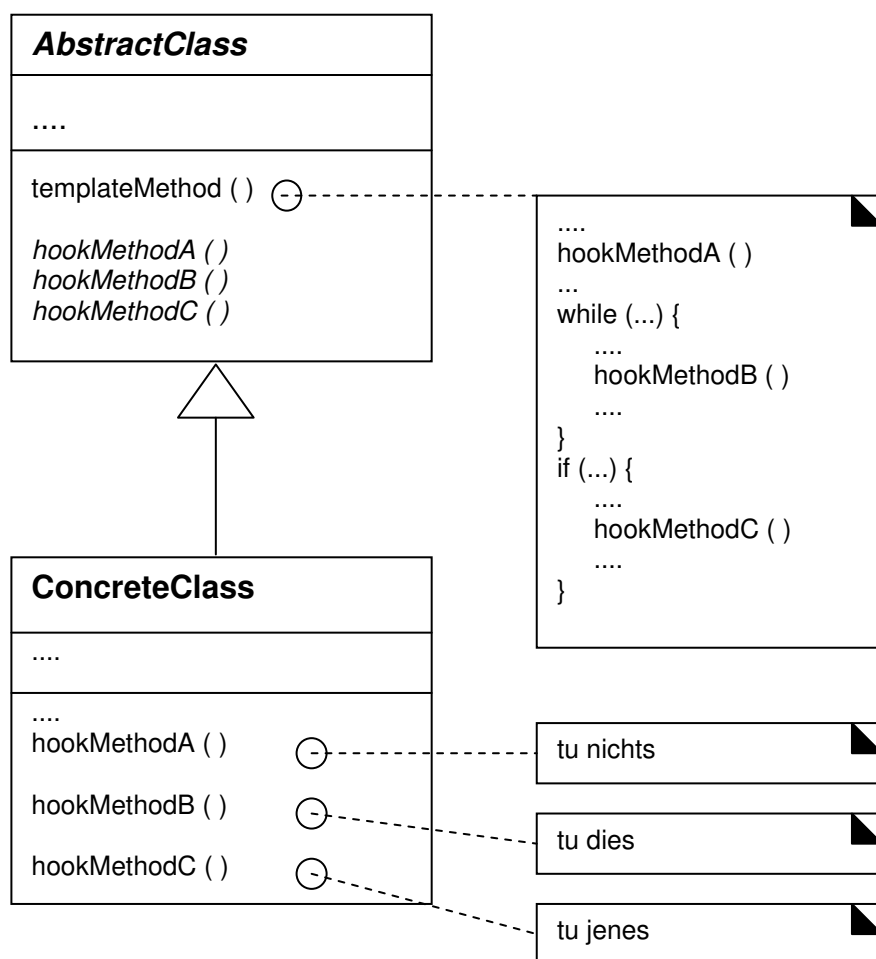
Ein kleiner grafischer Editor mit Memento-Funktionalität:



Der Quellcode soll hier nicht weiter dargestellt werden – er ist recht umfangreich...

4.7 Template Method

"Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern."
(Gamma, 327)



4.7.1 Problem

Angenommen, man benötigt ein Programm, welches den Inhalt einer Textdatei ausgibt. Das Problem ist natürlich relativ einfach zu lösen:

```
package appl;
// ...
public class Application1 {

    public static void main(String[] args) {
        final String filename = "src/appl/Application1.java";
        try (InputStreamReader reader = new InputStreamReader(
            new FileInputStream(filename))) {
            int ch = reader.read();
            System.out.println("Jetzt geht's los");
            while (ch != -1) {
                System.out.print((char)ch);
                ch = reader.read();
            }
            System.out.println("Das war's dann");
        }
        catch (final Exception e) {
            System.err.println(e);
        }
    }
}
```

Angenommen, man benötigt morgen ein weiteres Programm, welches die Anzahl der Zeichen einer Textdatei ermittelt. Auch dieses Problem ist schnell gelöst, zumal man ja bereits eine Vorlage besitzt. Das folgende Programm wird also per copy & paste "geschrieben":

```
package appl;
// ...
public class Application2 {

    public static void main(String[] args) {
        final String filename = "src/appl/Application1.java";
        try (InputStreamReader reader = new InputStreamReader(
            new FileInputStream(filename))) {
            int ch = reader.read();
            int count = 0;
            while (ch != -1) {
                count++;
                ch = reader.read();
            }
            System.out.println(count + " lines");
        }
        catch (final Exception e) {
            System.err.println(e);
        }
    }
}
```

Und morgen schreiben wir eine Anwendung, welche die Anzahl der Zeilen einer Datei ermittelt; und übermorgen eine Anwendung, welche die Anzahl der Wörter einer Datei ermittelt etc...

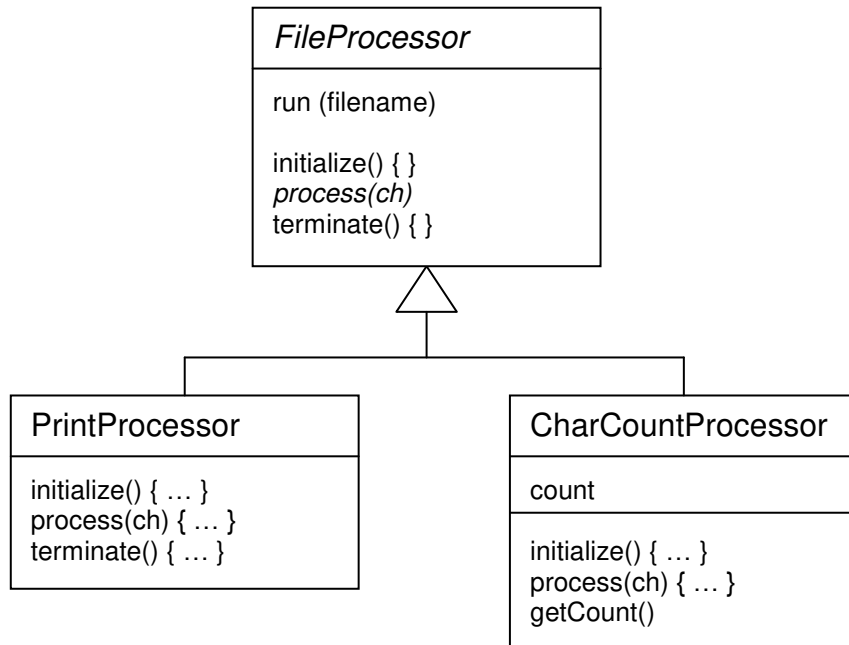
All diese Programme benutzen im Grunde ein und dasselbe Verfahren: man benötigt einen `FileInputStream`, auf den mittels eines `InputStreamReader` zugegriffen wird. Überall muss das erste Zeichen gelesen werden. Überall ist dann eine Schleife erforderlich, an deren Ende das jeweils nächste Zeichen gelesen werden muss. Und überall muss ein `Exception`-Handling implementiert werden. Dies ist die allgemeine Form der obigen Programme.

Die Programme unterscheiden sich nur exakt an drei Stellen. Diese drei Stellen legen die Vorverarbeitung, die Verarbeitung eines jeden Zeichens und die Nachverarbeitung fest. Sie legen den konkreten Inhalt der Anwendung fest.

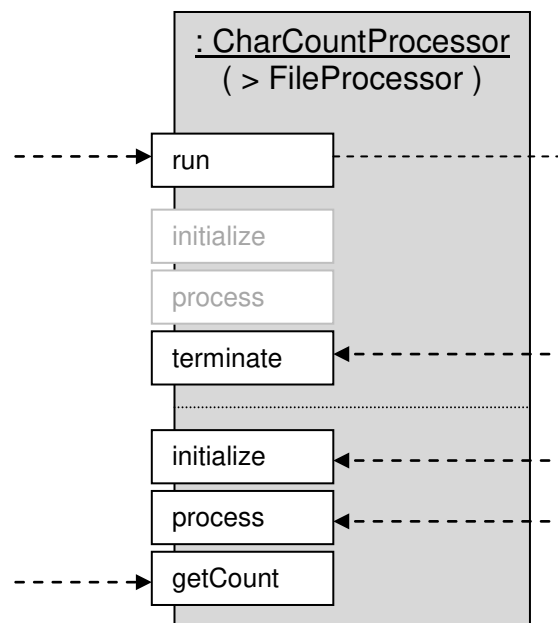
Es wäre schön, wenn die allgemeine Form ein einziges Mal abstrakt implementiert werden könnte – in Form einer Basisklasse. Die konkrete inhaltliche Seite eines Problems könnte dann in einer abgeleiteten Klasse implementiert werden.

4.7.2 Lösung

Das Klassendiagramm:



Ein Objektdiagramm:



Hier die Implementierung der allgemeinen Form:

```
package appl;
// ...
public abstract class FileProcessor {

    final public void run(String filename) {
        try (InputStreamReader reader= new InputStreamReader(
            new FileInputStream(filename))) {
            int ch = reader.read();
            this.initialize();
            while (ch != -1) {
                this.process((char) ch);
                ch = reader.read();
            }
            this.terminate();
        }
        catch(final Exception e) {
            System.out.println(e);
        }
    }

    protected void initialize() {
    }

    protected abstract void process(char ch);

    protected void terminate() {
    }
}
```

Die Klasse `FileProcessor` implementiert eine nicht überschreibbare, öffentliche Methode namens `run`. Diese Methode implementiert einen allgemeinen Algorithmus zur sequentiellen Verarbeitung der Zeichen einer Textdatei.

Die `run`-Methode ruft zu Beginn die Methode `initialize` auf – eine Methode, die in der Klasse `FileProcessor` leer implementiert ist (`protected`). Jedes gelesene Zeichen wird zur Verarbeitung an die Methode `process` delegiert – diese Methode ist in der Klasse `FileProcessor` als `abstract` definiert. Und am Ende der Verarbeitung wird die Methode `terminate` aufgerufen, welche analog zur `initialize`-Methode leer implementiert ist.

Die `run`-Methode wird als Template Method (Schablonenemethode) bezeichnet; die `initialize`-, `process`- und `terminate`-Methoden heißen Hook-Methods (Einschubmethoden).

Überall dort also, wo die `run`-Methode "nicht weiter weiß" (also nicht weiß, was nun konkret zu geschehen hat), wird an andere Methoden delegiert – an Methoden, die entweder leer implementiert oder als `abstract` definiert sind. Und diese Methoden können bzw. müssen dann von Methoden einer abgeleiteten Klasse geeignet überschrieben bzw. implementiert werden.

Z.B. von den Methoden einer Klasse `PrintProcessor`:

```
package appl;

public class PrintProcessor extends FileProcessor {
    @Override
    protected void initialize() {
        System.out.println("Jetzt geht's los");
    }
    @Override
    protected void process(char ch) {
        System.out.print(ch);
    }
    @Override
    protected void terminate() {
        System.out.println("Das war's dann");
    }
}
```

Während die Basisklasse `FileProcessor` die allgemeine Form einer bestimmten Gruppe von Anwendungen implementiert, enthält `PrintProcessor` den konkreten Inhalt der Anwendung.

Hier eine Klasse `CharCountProcessor`:

```
package appl;

public class CharCountProcessor extends FileProcessor {
    private int count;
    public int getCount() {
        return this.count;
    }
    @Override
    protected void initialize() {
        this.count = 0;
    }
    @Override
    protected void process(char ch) {
        this.count++;
    }
}
```

Analog könnten auch Klassen wie `LineCountProcessor` und `WordCountProcessor` implementiert werden.

Die Anwendung kann dann einen `PrintProcessor` und einen `CharCountProcessor` erzeugen und auf dieses Objekt jeweils deren `run`-Methode aufrufen:

```
package appl;

public class Application {

    public static void main(String[] args) {
        final String filename = "src/appl/Application.java";

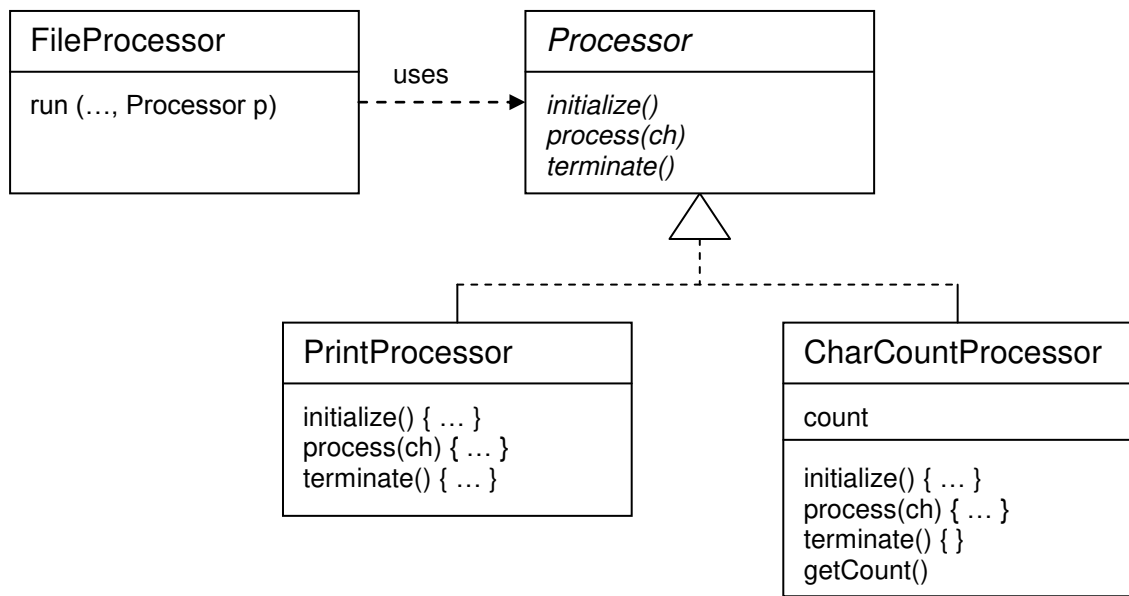
        new PrintProcessor().run(filename);

        final CharCountProcessor ccp = new CharCountProcessor();
        ccp.run(filename);
        System.out.println(ccp.getCount() + " characters");
    }
}
```

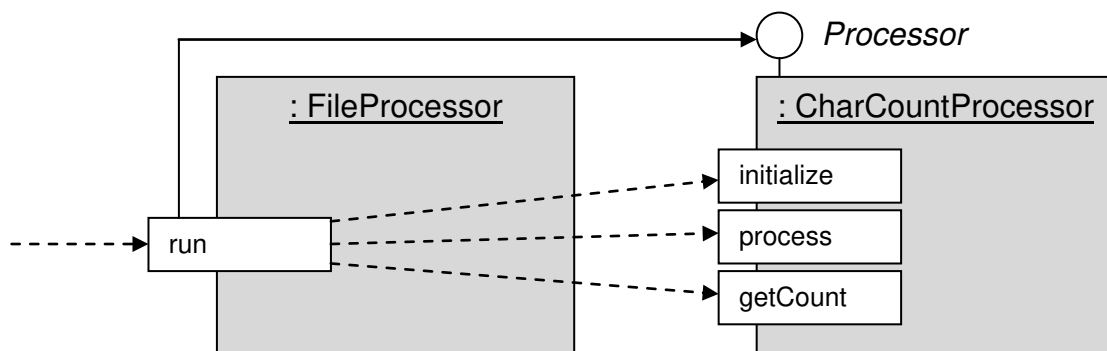
4.7.3 Delegation via Interface

Statt an Methoden einer abgeleiteten Klasse zu delegieren (statt also klassische Vererbung zu nutzen), könnten wir natürlich auch mittels eines Interfaces an ein anderes Objekt delegieren.

Das Klassendiagramm:



Ein Objektdiagramm:



Wir definieren ein `Processor`-Interface:

```

package appl;

public interface Processor {
    public abstract void initialize();
    public abstract void process(char ch);
    public abstract void terminate();
}
  
```

Hier ein `PrintProcessor`:

```
package appl;

public class PrintProcessor implements Processor {
    @Override
    public void initialize() {
        System.out.println("Jetzt geht's los");
    }
    @Override
    public void process(char ch) {
        System.out.print(ch);
    }
    @Override
    public void terminate() {
        System.out.println("Das war's dann");
    }
}
```

Und hier ein `CharCountProcessor`:

```
package appl;

public class CharCountProcessor implements Processor {
    private int count;
    public int getCount() {
        return this.count;
    }
    @Override
    public void initialize() {
        this.count = 0;
    }
    @Override
    public void process(char ch) {
        this.count++;
    }
    @Override
    public void terminate() {
        // do nothing
    }
}
```


Die Klasse `FileProcessor` kann nun als instanziiierbare Klasse implementiert werden. Sie enthält keine Hook-Methoden. Stattdessen wird der `run`-Methode eine `Processor`-Referenz übergeben, über welche sie dann die entsprechenden Methoden aufrufen kann:

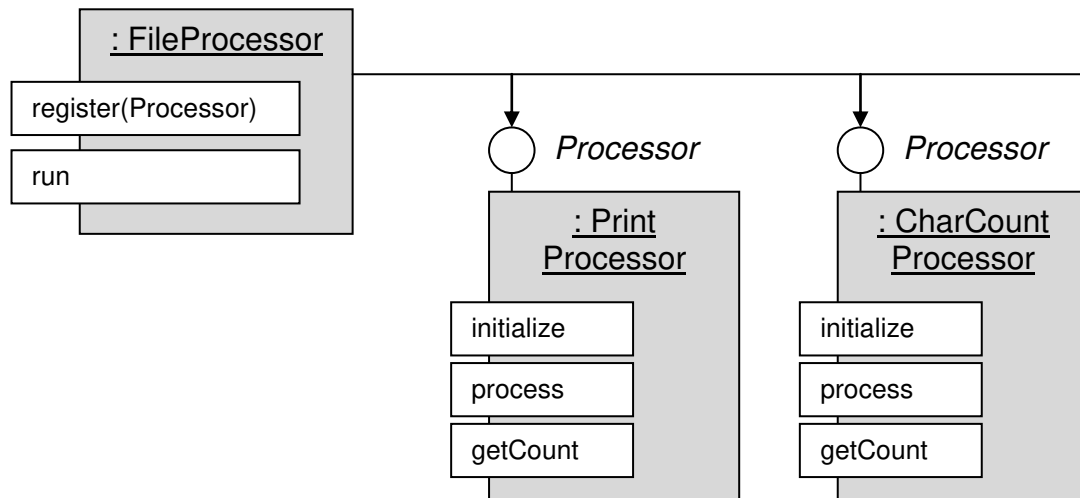
```
package appl;
// ...
public class FileProcessor {
    final public void run(String filename, Processor processor) {
        try (InputStreamReader reader= new InputStreamReader(
            new FileInputStream(filename))) {
            int ch = reader.read();
            processor.initialize();
            while (ch != -1) {
                processor.process((char) ch);
                ch = reader.read();
            }
            processor.terminate();
        }
        catch(final Exception e) {
            System.out.println(e);
        }
    }
}
```

Und hier die Test-Anwendung:

```
package appl;

public class Application {
    public static void main(String[] args) {
        final String filename = "src/appl/Application.java";
        final FileProcessor fp = new FileProcessor();
        fp.run(filename, new PrintProcessor());
        final CharCountProcessor ccp = new CharCountProcessor();
        fp.run(filename, ccp);
        System.out.println(ccp.getCount() + " characters");
    }
}
```

4.7.4 Erweiterung der Delegation



In der letzten Variante delegiert ein `FileProcessor` an genau einen einzigen `Processor`. Natürlich kann man die Klasse `FileProcessor` derart erweitern, dass bei einem Objekt dieser Klasse beliebig viele `Processor`-kompatible Objekte registriert werden können – die `run`-Methode muss dann die entsprechenden Methoden bei allen registrierten `Processor`-Objekt aufrufen:

```

package appl;
// ...
public class FileProcessor {

    private final List<Processor> processors = new ArrayList<>();

    public void register(Processor processor) {
        this.processors.add(processor);
    }

    final public void run(String filename) {
        try (final InputStreamReader reader= new InputStreamReader(
            new FileInputStream(filename))) {
            int ch = reader.read();
            this.processors.forEach(p -> p.initialize());
            while (ch != -1) {
                final int c = ch;
                this.processors.forEach(p -> p.process((char)c));
                ch = reader.read();
            }
            this.processors.forEach(p -> p.terminate());
        }
        catch (final Exception e) {
            System.out.println(e);
        }
    }
}
  
```

Dann reicht ein einziger `run`-Aufruf aus, um sowohl die Datei auszugeben als auch die Anzahl der Zeichen dieser Datei zu ermitteln:

```
package appl;

public class Application {

    public static void main(String[] args) {
        final String filename = "src/appl/Application.java";

        final FileProcessor fp = new FileProcessor();

        final CharCountProcessor ccp = new CharCountProcessor();

        fp.register(new PrintProcessor());
        fp.register(ccp);

        fp.run(filename);

        System.out.println(ccp.getCount() + " characters");
    }
}
```

4.7.5 Ein Gruppenwechsel-Prozessor

Gegeben sei ein Stapel Lochkarten, auf denen jeweils eine Kunden-Nummer und ein Umsatz gespeichert sind. Diese Lochkarten sind derart sortiert, dass die Umsatz-Karten eines Kunden jeweils eine Gruppe bilden:

```
1000 500
1000 300
1000 100
2000 500
3000 100
3000 200
```

Man erkennt drei Gruppen: die Umsatz-Gruppe für den Kunden 1000, die Gruppe für den Kunden 2000 und die Gruppe für den Kunden 3000. Die in dem Lochkartenstapel nur implizit enthaltene Gruppenstruktur soll nun explizit gemacht werden. Der hierfür erforderliche Algorithmus wird als Gruppenwechsel bezeichnet. Das Resultat der Verarbeitung könnte z.B. eine Druckliste sein, welche zusätzlich zu den Eingabedaten auch noch die Gruppensummen und die Gesamtsumme aller Umsätze beinhaltet:

```
Umsaetze
1000
    500.0
    300.0
    100.0
    ----
    900.0
2000
    500.0
    ----
    500.0
3000
    100.0
    200.0
    ----
    300.0
=====
1700.0
```

Natürlich gibt es Gruppenwechsel ganz verschiedenen konkreten Inhalts. Es wäre also auch hier sinnvoll, den abstrakten Mechanismus von den konkreten Inhalten (und von der konkreten Form der Ausgabe) zu trennen.

Die abstrakte Form lässt sich wie folgt beschreiben:

- Am Anfang muss eine Anfangsverarbeitung stattfinden.
- Für jede Gruppe muss eine Anfangsverarbeitung stattfinden.
- Für jede Lochkarte muss eine Positionsverarbeitung stattfinden.
- Für jede Gruppe muss eine Endeverarbeitung stattfinden.
- Am Ende muss eine Endeverarbeitung stattfinden.

Hier die Implementierung dieser abstrakten Form:

```
package appl;

import java.util.Iterator;

public abstract class GroupProcessor<T> {

    private T read(Iterator<T> iterator) {
        return iterator.hasNext() ? iterator.next() : null;
    }

    public final void run(Iterator<T> iterator) {
        this.handleBegin();
        T old = null;
        T current = this.read(iterator);
        while (current != null) {
            this.handleGroupBegin(current);
            old = current;
            while (current != null && !this.isNewGroup(old, current)) {
                this.handlePosition(current);
                current = this.read(iterator);
            }
            this.handleGroupEnd(current);
        }
        this.handleEnd();
    }

    protected abstract boolean isNewGroup(T oldObject, T newObject);

    protected abstract void handleBegin();

    protected abstract void handleGroupBegin(T object);

    protected abstract void handlePosition(T object);

    protected abstract void handleGroupEnd(T object);

    protected abstract void handleEnd();
}
```

GroupProcessor ist eine abstrakte Basisklasse (für Lochkarten des Typs T). Die run-Methode ist die Template-Method, die den abstrakten Mechanismus des Gruppenwechsels implementiert und hierzu an wohl-definierten Stellen Hook-Methoden aufruft (handleBegin, handleGroupBegin etc.). Zusätzlich existiert eine Hook-Methode namens is-

`NewGroup`, die `true` zurückliefert muss, wenn die beiden ihr übergebenen Lochkarten zu verschiedenen Gruppen gehören. Die `run`-Methode geht davon aus, dass ihr ein `Iterator<T>` übergeben wird, mittels dessen über den Eingabe-Stapel iteriert werden kann.

Im folgenden eine konkrete Anwendung.

Die Lockkarten seien vom Typ `Umsatz`:

```
package appl;

public class Umsatz {

    private final int kdNr;
    private final double umsatz;

    public Umsatz(int kdNr, double umsatz) {
        this.kdNr = kdNr;
        this.umsatz = umsatz;
    }

    public int getKdNr() { return this.kdNr; }
    public double getUmsatz() { return this.umsatz; }

    @Override
    public String toString() {
        return this.kdNr + " " + this.umsatz;
    }
}
```

Dann kann folgende `UmsatzProcessor`-Klasse implementiert werden:

```
package appl;

class UmsatzProcessor extends GroupProcessor<Umsatz> {

    private double totalSum;
    private double groupSum;

    @Override
    protected boolean isNewGroup(Umsatz old, Umsatz current) {
        return old.getKdNr() != current.getKdNr();
    }

    @Override
    protected void handleBegin() {
        System.out.println("Umsaetze");
        this.totalSum = 0;
    }

    @Override
    protected void handleGroupBegin(Umsatz object) {
        System.out.println("\t" + object.getKdNr());
        this.groupSum = 0;
    }

    @Override
    protected void handlePosition(Umsatz object) {
        System.out.println("\t\t" + object.getUmsatz());
    }
}
```

```
        this.groupSum += object.getUmsatz();
    }

    @Override
    protected void handleGroupEnd(Umsatz object) {
        System.out.println("\t\t-----");
        System.out.println("\t\t" + this.groupSum);
        this.totalSum += this.groupSum;
    }

    @Override
    protected void handleEnd() {
        System.out.println("\t\t=====");
        System.out.println("\t\t" + this.totalSum);
    }
}
```

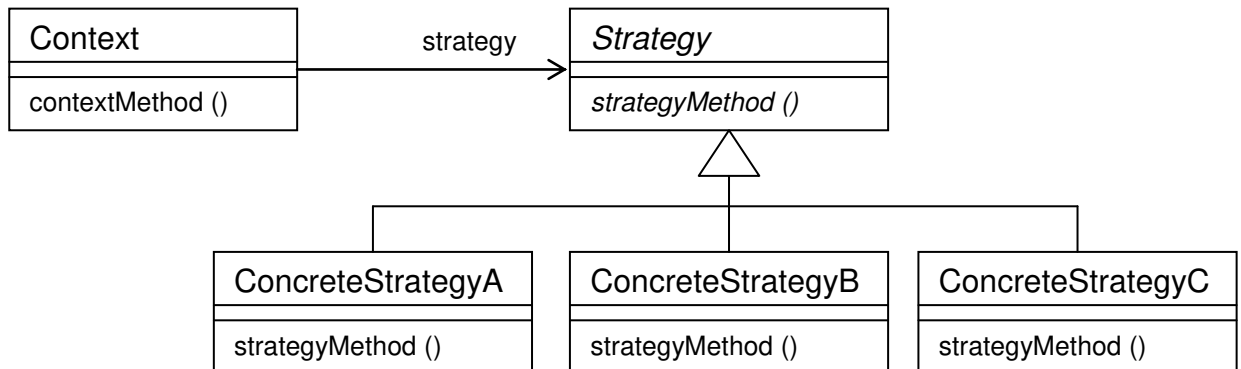
Diese Klasse enthält die Implementierung der Hook-Methoden. Sie implementiert also den konkreten Inhalt der Anwendung.

Die folgende Anwendung produziert dann die oben dargestellten Ausgaben:

```
package appl;
// ...
public class Application {
    public static void main(String[] args) throws Exception {
        final ArrayList<Umsatz> list = new ArrayList<>();
        list.add(new Umsatz(1000, 500));
        list.add(new Umsatz(1000, 300));
        list.add(new Umsatz(1000, 100));
        list.add(new Umsatz(2000, 500));
        list.add(new Umsatz(3000, 100));
        list.add(new Umsatz(3000, 200));
        new UmsatzProcessor().run(list.iterator());
    }
}
```

4.8 Strategy

"Definiere eine Familie von Algorithmen, kapsle jeden einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu ändern." (Gamma, 333)



4.8.1 Problem

Gegeben sei eine Klasse `SimpleArrayList` (wie sie auch bereits im Iterator-Abschnitt verwendet wurde):

```
package util;

public class SimpleArrayList<T> {

    private T[] elements = this.createArray(2);
    private int size;

    public void add(T element) {
        this.ensureCapacity();
        this.elements[this.size] = element;
        this.size++;
    }

    public int size() {
        return this.size;
    }

    public T get(int index) {
        if (index < 0 || index >= this.size)
            throw new IndexOutOfBoundsException();
        return this.elements[index];
    }

    private void ensureCapacity() {
        if (this.size == this.elements.length) {
            final T[] newElements = this.createArray(2 * this.size);
            for (int i = 0; i < this.size; i++)
                newElements[i] = this.elements[i];
            this.elements = newElements;
        }
    }

    @SuppressWarnings("unchecked")
    private T[] createArray(int length) {
        return (T[]) new Object[length];
    }
}
```

Die Klasse benutzt einen Array, welcher bei Bedarf durch einen neuen Array ersetzt wird (wobei die Elemente des alten Arrays in den neuen Array kopiert werden). Der neue Array ist stets doppelt so groß wie der alte (siehe `ensureCapacity`).

Hier eine Test-Anwendung:

```
package appl;

import util.SimpleArrayList;

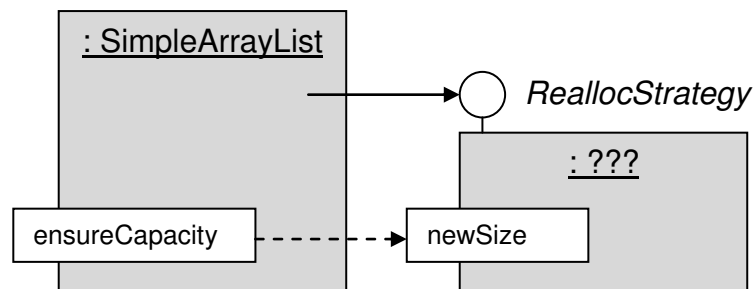
public class Application {

    public static void main(String[] args) {
        final SimpleArrayList<String> list = new SimpleArrayList<>();
        list.add("spring");
        list.add("summer");
        list.add("autumn");
        list.add("winter");
        for (int i = 0; i < list.size(); i++)
            System.out.println(list.get(i));
    }
}
```

Da das Array-Objekt die Initialgröße 2 hat, findet bei dieser Anwendung genau eine Reallokation statt (aus 2 mach 4...).

Manchmal wäre es sicher wünschenswert, eine bestimmte Reallokations-Strategie einstellen zu können (statt die Größe jeweils zu verdoppeln, könnte man z.B. eine lineare Strategie wählen – bei jeder Reallokation würde ein neuer Array erzeugt, der um einen konstanten Wert größer ist als der alte).

4.8.2 Lösung



Man definiert ein Interface, dessen Implementierungen Reallokations-Strategien enthalten:

```
package util;

@FunctionalInterface
public interface ReallocStrategy {
    public abstract int newSize(int oldSize);
}
```

Mittels einer Reallokations-Strategie kann somit die neue Größe bei einer Reallokation bestimmt werden (aufgrund der bisherigen Größe).

Die Klasse SimpleArrayList wird wie folgt erweitert:

```
package util;

public class SimpleArrayList<T> {

    private T[] elements;
    private int size;
    private final ReallocStrategy strategy;

    private static final ReallocStrategy defaultStrategy =
        new ReallocStrategy() {
            @Override
            public int newSize(int oldSize) {
                return oldSize * 2;
            }
        };

    public SimpleArrayList(int size, ReallocStrategy strategy) {
        this.elements = this.createArray(size);
        this.strategy = strategy;
    }

    public SimpleArrayList() {
        this(2, defaultStrategy);
    }

    public void add(T element) { ... }
    public int size() { ... }
    public T get(int index) { ... }
```

```
private void ensureCapacity() {  
    if (this.size == this.elements.length) {  
        final T[] newElements =  
            this.createArray(this.strategy.newSize(this.size));  
        for (int i = 0; i < this.size; i++)  
            newElements[i] = this.elements[i];  
        this.elements = newElements;  
    }  
}  
  
private T[] createArray(int length) { ... }  
}
```

Sofern der parameterlose Konstruktor genutzt wird, wird ein Array mit der Anfangsgröße 2 erzeugt und eine `defaultStrategy` verwendet (Verdopplungs-Strategie). Dem zweiten Konstruktor wird eine explizite Anfangsgröße und eine explizite Strategie übergeben. (Man beachte den Aufruf von `newSize` in der Methode `ensureCapacity`.)

Die folgende Anwendung zeigt die Benutzung des zweiten Konstruktors:

```
package appl;  
  
import util.ReallocStrategy;  
import util.SimpleArrayList;  
  
public class Application {  
    public static void main(String[] args) {  
        final SimpleArrayList<String> list =  
            new SimpleArrayList<>(0, new ReallocStrategy() {  
                @Override  
                public int newSize(int oldSize) {  
                    return oldSize + 1;  
                }  
            });  
        list.add("spring");  
        list.add("summer");  
        list.add("autumn");  
        list.add("winter");  
        for (int i = 0; i < list.size(); i++)  
            System.out.println(list.get(i));  
    }  
}
```

Es wird hier offenbar linear allokiert (das Array der `SimpleArrayList` wird jeweils um ein weiteres Element vergrößert – nicht unbedingt intelligent...)

4.8.3 Lambdas

Statt die Strategy in Form anonymer Klassen zu implementieren, können auch Lambdas verwendet werden:

```
package util;

public class SimpleArrayList<T> {

    // ...

    private static final ReallocStrategy defaultStrategy =
        oldSize -> oldSize * 2;

    // ...

}

package appl;

import util.SimpleArrayList;

public class Application {

    public static void main(String[] args) {
        final SimpleArrayList<String> list =
            new SimpleArrayList<>(0, oldSize -> oldSize + 1);

        // ...

    }

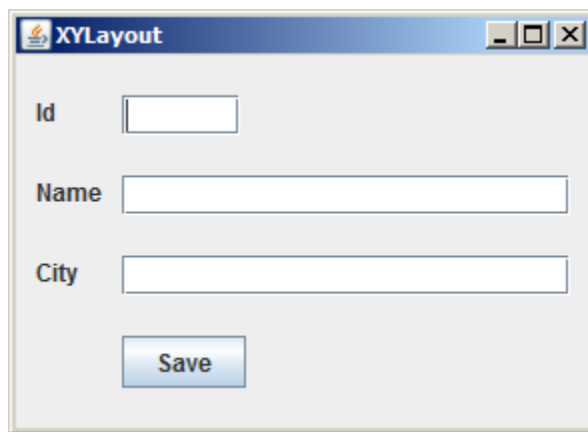
}
```

4.8.4 XYLayout

Das AWT ermöglicht bekanntlich die Positionierung und Dimensionierung der Komponenten mittels `LayoutManager`n. `LayoutManager` ist ein Interface, welches z.B. in den Standardklassen `FlowLayout`, `GridBagLayout`, `GridLayout` und `BorderLayout` implementiert ist

Diese Klassen implementieren eine "Layout-Strategie".

Im folgenden wird eine weitere einfache Layout-Strategie vorgestellt: die `XYLayout`-Strategie. Es handelt sich um eine vereinfachte Form der recht schwer zu benutzenden `GridBagLayout`-Strategie.



Die Panel-Klasse ist wie folgt implementiert:

```
package appl;
// ...
import util.XYConstraint;
import util.XYLayout;

public class TestPanel extends JPanel {

    private final JTextField textFieldId = new JTextField(5);
    private final JTextField textFieldName = new JTextField(20);
    private final JTextField textFieldCity = new JTextField(20);
    private final JButton buttonSave = new JButton("Save");

    public TestPanel() {
        this.setLayout(new XYLayout(10, 20));
        this.add(new JLabel("Id"), new XYConstraint(0, 0));
        this.add(this.textFieldId, new XYConstraint(1, 0));
        this.add(new JLabel("Name"), new XYConstraint(0, 1));
        this.add(this.textFieldName, new XYConstraint(1, 1));
        this.add(new JLabel("City"), new XYConstraint(0, 2));
        this.add(this.textFieldCity, new XYConstraint(1, 2));
        this.add(this.buttonSave, new XYConstraint(1, 3));
    }
}
```

Ein `XYConstraint` repräsentiert die "logische" Position einer Komponente:

```
package util;

public class XYConstraint {
    public final int x;
    public final int y;

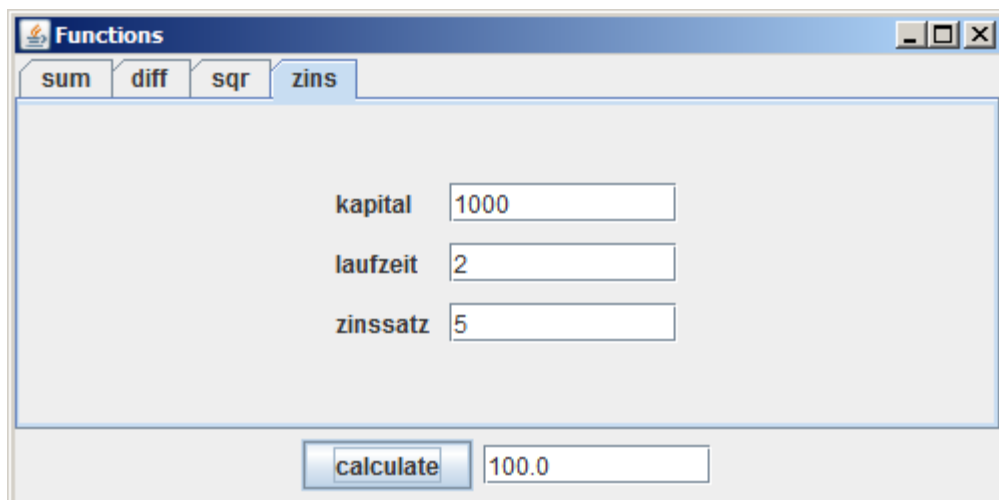
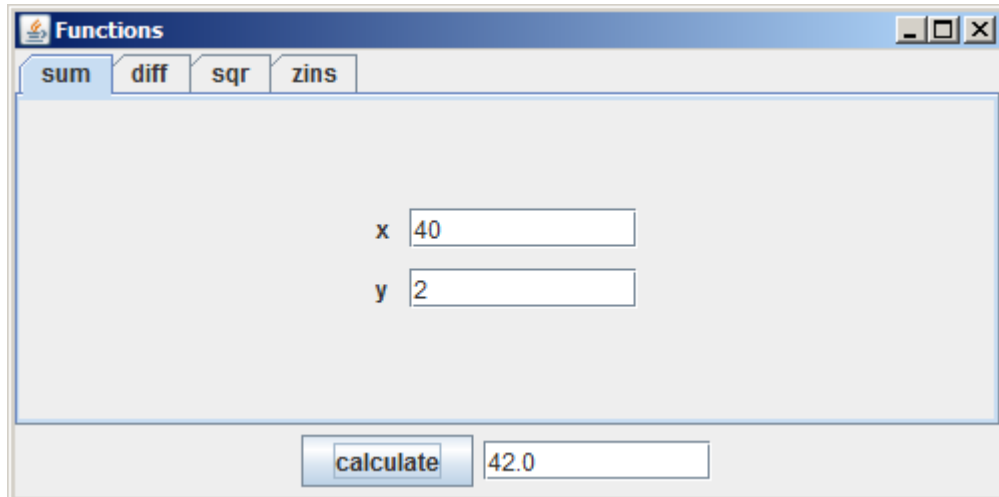
    public XYConstraint(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Die Klasse `XYLayout` ist nicht gerade einfach. Was die genaue Implementierung angeht, sei auf den Quelltext verwiesen.

4.8.5 Functions

Die folgende Anwendung implementiert eine Oberfläche, die einerseits an Strategy-Objekte delegiert und deren Aufbau andererseits auch durch diese Strategy-Objekte bestimmt wird. Es handelt sich dabei um unterschiedliche "Berechnungs-Strategien".

Die Anwendung präsentiert sich dem Benutzer etwa wie folgt:



Die gesamte Anwendung wird über folgendes Array gesteuert (in der Klasse Application):

```
final Function[] functions = new Function[] {  
    new SumFunction(),  
    new DiffFunction(),  
    new SqrFunction(),  
    new ZinsFunction()  
};
```


Alle Klassen, die hier instanziiert werden, implementieren das Interface `Function`:

```
package core;

public interface Function {
    public abstract String getName();
    public abstract int getArgumentCount();
    public abstract String getArgumentName(int index);
    public abstract double evaluate(double[] arguments);
}
```

Eine Funktion hat einen Namen. Sie hat eine bestimmte Anzahl von Argumenten. Jedes Argument hat ebenfalls einen Namen. Und eine Funktion muss evaluiert werden können.

Um die Implementierung konkreter `Function`-Klassen zu vereinfachen, existiert folgende abstrakte Klasse, die das Interface bereits partiell implementiert:

```
package core;

public abstract class AbstractFunction implements Function {
    private final String name;
    private final String[] argumentNames;

    public AbstractFunction(String name, String... argumentNames) {
        this.name = name;
        this.argumentNames = argumentNames.clone();
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public int getArgumentCount() {
        return this.argumentNames.length;
    }

    @Override
    public String getArgumentName(int index) {
        return this.argumentNames[index];
    }
}
```

Hier einige instanziiierbare Funktions-Klassen:

```
package functions;  
  
import core.AbstractFunction;  
  
public class SumFunction extends AbstractFunction {  
    public SumFunction() {  
        super("sum", "x", "y");  
    }  
    @Override  
    public double evaluate(double[] arguments) {  
        return arguments[0] + arguments[1];  
    }  
}
```

```
package functions;  
  
import core.AbstractFunction;  
  
public class SqrFunction extends AbstractFunction {  
    public SqrFunction() {  
        super("sqr", "value");  
    }  
    @Override  
    public double evaluate(double[] arguments) {  
        return arguments[0] * arguments[0];  
    }  
}
```

```
package functions;  
  
import core.AbstractFunction;  
  
public class ZinsFunction extends AbstractFunction {  
    public ZinsFunction() {  
        super("zins", "kapital", "laufzeit", "zinssatz");  
    }  
  
    @Override  
    public double evaluate(double[] arguments) {  
        return arguments[0] * arguments[1] * arguments[2] / 100;  
    }  
}
```

Die Klasse `FunctionsPanel` beliebige und beliebig viele Functions anbieten:

```
package appl;
// ...
import core.Function;

public class FunctionsPanel extends JPanel {

    private final JTabbedPane tabbedPane = new JTabbedPane();
    private final JPanel controlPanel = new JPanel();
    private final JButton calcButton = new JButton("calculate");
    private final JTextField resultTextField = new JTextField(10);
    private final Function[] functions;

    public FunctionsPanel(Function[] functions) {
        this.functions = functions;
        this.setLayout(new BorderLayout());
        this.add(this.tabbedPane, BorderLayout.CENTER);
        this.add(this.controlPanel, BorderLayout.SOUTH);
        this.controlPanel.setLayout(new FlowLayout());
        this.controlPanel.add(this.calcButton);
        this.controlPanel.add(this.resultTextField);
        this.calcButton.addActionListener(e -> this.onCalculate());
        for (final Function function : functions) {
            final JPanel panel = this.createFunctionPanel(function);
            this.tabbedPane.add(function.getName(), panel);
        }
    }

    private JPanel createFunctionPanel(Function function) {
        final JPanel panel = new JPanel();
        panel.setLayout(new GridBagLayout());
        final GridBagConstraints c = new GridBagConstraints();
        c.insets = new Insets(5, 5, 5, 5);
        c.anchor = GridBagConstraints.WEST;
        for (int i = 0; i < function.getArgumentCount(); i++) {
            c.gridy = i;
            c.gridx = 0;
            panel.add(new JLabel(function.getArgumentName(i)), c);
            c.gridx = 1;
            panel.add(new JTextField(10), c);
        }
        return panel;
    }

    private void onCalculate() {
        final int tab = this.tabbedPane.getSelectedIndex();
        final JPanel panel = (JPanel)this.tabbedPane.getComponentAt(tab);
        final Function function = this.functions[tab];
        final int n = function.getArgumentCount();
        final double[] arguments = new double[n];
        for (int i = 0; i < n; i++) {
            final JTextField tf = (JTextField)panel.getComponent(i * 2 + 1);
            arguments[i] = Double.parseDouble(tf.getText());
        }
        final double result = function.evaluate(arguments);
        this.resultTextField.setText(String.valueOf(result));
    }
}
```

Die Application startet den FunctionsPanel und übergibt dabei zu präsentierenden Functions:

```
package appl;
// ...
import core.Function;

public class Application {
    public static void main(String[] args) throws Exception {
        final JFrame frame = new JFrame("Functions");
        final Function[] functions = new Function[] {
            new SumFunction(),
            new DiffFunction(),
            new SqrFunction(),
            new ZinsFunction()
        };
        frame.add(new FunctionsPanel(functions));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setBounds(100, 100, 500, 250);
        frame.setVisible(true);
    }
}
```

4.8.6 Circuits

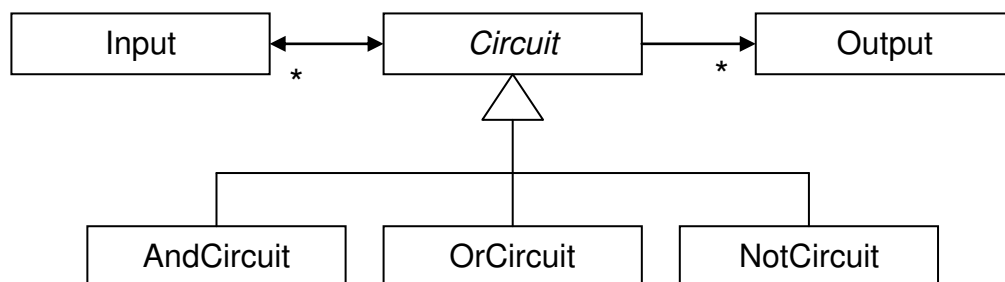
Man möchte ein GUI-Programm schreiben, mittels dessen logische Schaltungen erstellt werden können.

Der Benutzer kann And-, Or- und Not-Schaltungen platzieren. Eine And-Schaltung z.B. hat zwei Eingänge und einen Ausgang. (Später wird es aber vielleicht auch Schaltungen mit mehr als zwei Eingängen und mehr als einem Ausgang geben – z.B. einen Halbaddierer).

Der Benutzer möchte die Eingänge "toggeln" können – wobei der Ausgang (die Ausgänge) dann natürlich jeweils neu berechnet werden müssen.

Der Benutzer möchte zusätzlich die Schaltungen natürlich miteinander "verdrahten" können – also jeweils einen Ausgang einer Schaltung mit einem (bislang noch "freien") Eingang einer anderen Schaltung verbinden können. Wird dann ein Eingang getoggelt, muss sich die Berechnung natürlich rekursiv über alle mit den Ausgängen der aktuellen Schaltung verbundenen Schaltungen erstrecken.

Die erste Entwurfsidee ist sicherlich die, eine Klasse `Circuit` zu definieren, von welcher dann die Klassen `AndCircuit`, `OrCircuit`, `NotCircuit` etc. abgeleitet sind:

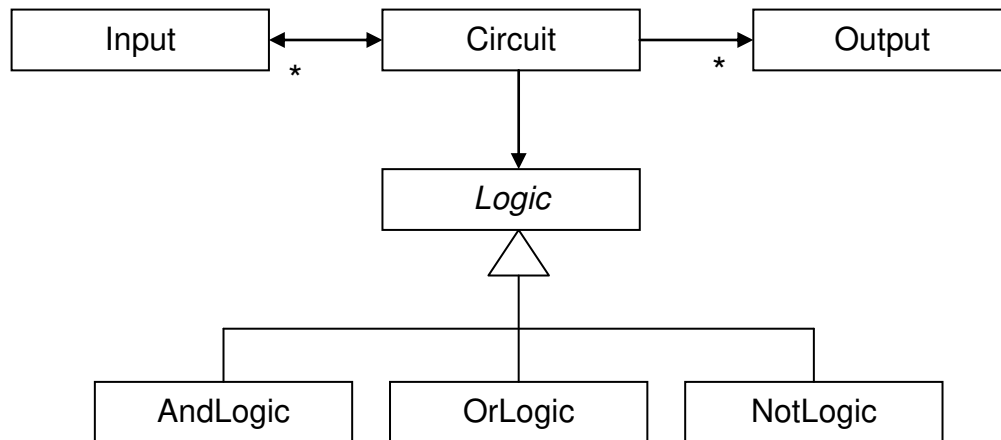


Was aber, wenn der Benutzer nun aus eine And-Schaltung eine Or-Schaltung machen möchte? Intern müsste man dann ein neues `OrCircuit`-Objekt erzeugen und alle Verbindungen, die das bisherige `AndCircuit`-Objekt zu anderen `Circuits` hatte, auf dieses neue Objekt umsetzen.

Weiterhin: die Verwaltung (und Erzeugung!) der `Input`- und `Output`-Objekte ist eine Sache – eine andere Sache ist die Logik, die von den verschiedenen Schaltungen ausgeführt wird.

Es bietet sich also an, für diese beiden Aspekte jeweils zwei verschiedene Objekte zu erzeugen – ein `Circuit`-Objekt und ein Objekt einer von `Logic` abgeleiteten Klasse. Dieses letzte Objekt implementiert dann die Berechnungsstrategie. Und soll bei einer Schaltung dann eine andere Berechnung hinterlegt werden, so muss nur das bisherige `Logic`-Objekt durch ein Objekt einer anderen Klasse ersetzt werden – das `Circuit`-Objekt kann (mitsamt seinen Verbindungen) bestehen bleiben.

Das Klassendiagramm:



Hier zunächst die Klassen des Kerns (Input, Output, Logic, Circuit):

```
package core;

public class Input {

    private boolean state;
    private final Circuit circuit;

    public Input(Circuit circuit) {
        this.circuit = circuit;
    }

    public void setState(boolean state) {
        this.state = state;
        this.circuit.calculate();
    }

    public boolean getState() {
        return this.state;
    }

}
```

```
package core;

public class Output {

    private boolean state;

    public void setState(boolean state) {
        this.state = state;
    }

    public boolean getState() {
        return this.state;
    }

}
```

```
package core;

public abstract class Logic {

    private final String label;
    private final int inputCount;
    private final int outputCount;

    public Logic(String label, int inputCount, int outputCount) {
        this.label = label;
        this.inputCount = inputCount;
        this.outputCount = outputCount;
    }

    public String getLabel() { return this.label; }
    public int getInputCount() { return this.inputCount; }
    public int getOutputCount() { return this.outputCount; }

    public abstract boolean[] calculate(boolean[] inputs);
}
```

```
package core;

public class Circuit {

    private Input[] inputs;
    private Output[] outputs;
    private Logic logic;

    public Circuit(Logic logic) {
        this.setLogic(logic);
    }

    public void calculate() {
        final boolean[] inputs = new
boolean[this.logic.getInputCount()];
        for (int i = 0; i < inputs.length; i++)
            inputs[i] = this.inputs[i].getState();
        final boolean[] outputs = this.logic.calculate(inputs);
        for (int i = 0; i < outputs.length; i++)
            this.outputs[i].setState(outputs[i]);
    }

    public Logic getLogic() {
        return this.logic;
    }

    public void setLogic(Logic logic) {
        this.logic = logic;
        this.inputs = new Input[logic.getInputCount()];
        for (int i = 0; i < this.inputs.length; i++)
            this.inputs[i] = new Input(this);
        this.outputs = new Output[logic.getOutputCount()];
        for (int i = 0; i < this.outputs.length; i++)
            this.outputs[i] = new Output();
        this.calculate();
    }
}
```

```
public Input getInput(int index) {
    return this.inputs[index];
}

public Output getOutput(int index) {
    return this.outputs[index];
}

@Override
public String toString() {
    String s = this.logic.getLabel() + "\t";
    for (int i = 0; i < this.logic.getInputCount(); i++) {
        if (i > 0)
            s += ",";
        s += this.inputs[i].getState();
    }
    s += " --> ";
    for (int i = 0; i < this.logic.getOutputCount(); i++) {
        if (i > 0)
            s += ",";
        s += this.outputs[i].getState();
    }
    return s;
}
}
```

Hier einige konkrete von Logic abgeleitete Klassen:

```
package logics;

import core.Logic;

public class AndLogic extends Logic {
    public AndLogic() {
        super("&&", 2, 1);
    }
    @Override
    public boolean[] calculate(boolean[] inputs) {
        return new boolean[] { inputs[0] && inputs[1] };
    }
}
```

```
package logics;

import core.Logic;

public class OrLogic extends Logic {
    public OrLogic() {
        super("||", 2, 1);
    }
    @Override
    public boolean[] calculate(boolean[] inputs) {
        return new boolean[] { inputs[0] || inputs[1] };
    }
}
```



```
package logics;

import core.Logic;

public class NotLogic extends Logic {
    public NotLogic() {
        super("!", 1, 1);
    }
    @Override
    public boolean[] calculate(boolean[] inputs) {
        return new boolean[] { ! inputs[0] };
    }
}
```

(Man beachte, dass diese Klassen auch als Singleton-Klassen hätten definiert werden können – was völlig unproblematisch wäre, weil ihre Objekte keinen Zustand besitzen.)

Hier die Test-Anwendung:

```
package appl;

import logics.AndLogic;
import logics.NotLogic;
import logics.OrLogic;
import core.Circuit;

public class Application {
    public static void main(String[] args) {

        final Circuit c1 = new Circuit(new AndLogic());
        final Circuit c2 = new Circuit(new OrLogic());
        final Circuit c3 = new Circuit(new NotLogic());
        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
        System.out.println();

        c1.getInput(0).setState(true);
        c1.getInput(1).setState(true);
        System.out.println(c1);
        System.out.println();

        c2.getInput(0).setState(true);
        System.out.println(c2);
        System.out.println();

        c3.getInput(0).setState(true);
        System.out.println(c3);
        System.out.println();
    }
}
```

Die Ausgaben des obigen Programms:

```
&&  false,false --> false
||   false,false --> false
!    false --> true

&&  true,true --> true

||   true,false --> true

!    true --> false
```

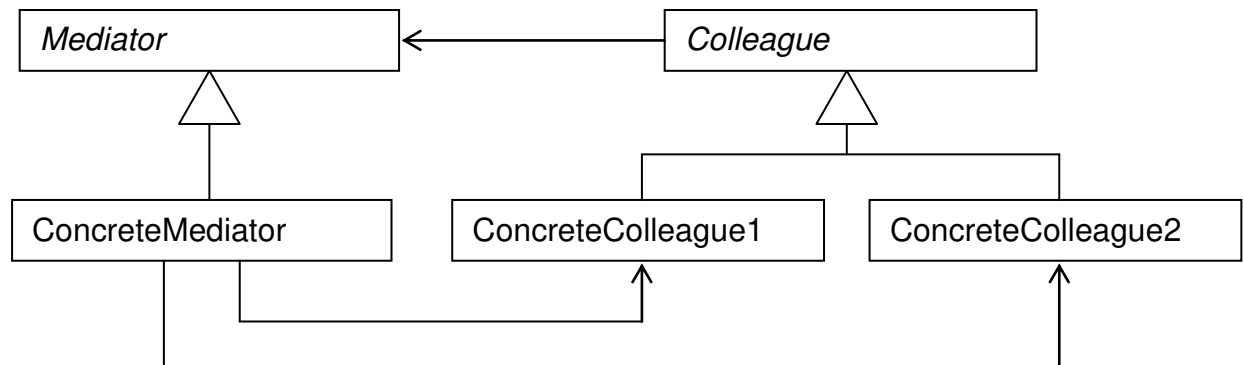
Aufgaben

Erweitern Sie die obige Implementierung um folgende Features:

Ein Eingang kann von genau einem Ausgang versorgt werden; umgekehrt kann ein Ausgang mehrere Eingänge versorgen. Ausgänge müssen mit Eingängen verknüpft werden können. Das Togglen eines Eingangs verändert dann nicht nur den Zustand einer Schaltung, sondern auch die Zustände all der Schaltungen, deren Eingänge direkt oder indirekt von der getoggelten Schaltung versorgt werden.

4.9 Mediator

"Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es Ihnen, das Zusammenspiel der Objekte von ihnen unabhängig zu variieren." (Gamma, 375)



4.9.1 Problem

Im folgenden wird die Situation eines Chatrooms mit seinen Teilnehmern modelliert.

Ein Teilnehmer kann allen anderen Teilnehmern eine Meldung zukommen lassen, welche von den anderen Teilnehmern empfangen werden muss:

```
package appl;

import java.util.ArrayList;
import java.util.List;

public class Participant {

    private final String name;
    private final List<Participant> participants = new ArrayList<>();

    public Participant(String name) {
        this.name = name;
    }

    public String getName() { return this.name; }

    public void add(Participant participant) {
        this.participants.add(participant);
    }

    public void send(String message) {
        System.out.println();
        System.out.printf("Sender %s sends message: %s\n",
                           this.name, message);
        System.out.println("-----");
        this.participants.forEach(p -> p.notify(this, message));
    }

    public void notify(Participant sender, String message) {
        System.out.printf(
            "User %s received message: %s (Sender = %s)\n",
            this.name, message, sender.name);
    }
}
```

Leider muss eine Anwendung nun jeden Teilnehmer mit jedem anderen verbinden:

```
package appl;

public class Application {

    public static void main(String[] args) {
        final Participant meier = new Participant("Meier");
        final Participant mueller = new Participant("Mueller");
        final Participant franke = new Participant("Franke");
        meier.add(mueller);
        meier.add(franke);
        mueller.add(meier);
        mueller.add(franke);
        franke.add(meier);
        franke.add(mueller);
        meier.send("Glueck");
        mueller.send("Glanz");
        franke.send("Ruhm");
    }
}
```

Bei drei Teilnehmer gibt's bereits sechs Verbindungen (allgemein: $n!$)
Besser wäre ein Objekt (ein `Chatroom`), welches zwischen den Teilnehmern vermittelt.

Die Ausgaben:

```
Sender Meier sends message: Glueck
-----
User Mueller received message: Glueck (Sender = Meier)
User Franke received message: Glueck (Sender = Meier)

Sender Mueller sends message: Glanz
-----
User Meier received message: Glanz (Sender = Mueller)
User Franke received message: Glanz (Sender = Mueller)

Sender Franke sends message: Ruhm
-----
User Meier received message: Ruhm (Sender = Franke)
User Mueller received message: Ruhm (Sender = Franke)
```

4.9.2 Lösung

Die Klasse Chatroom:

```
package appl;

import java.util.ArrayList;
import java.util.List;

public class Chatroom {
    private final List<Participant> participants = new ArrayList<>();

    public void add(Participant participant) {
        this.participants.add(participant);
    }

    public void remove(Participant participant) {
        this.participants.remove(participant);
    }

    public void send(final Participant sender, final String message) {
        System.out.println();
        System.out.printf("Sender %s sends message: %s\n",
            sender.getName(), message);
        System.out.println("-----");
        this.participants.forEach(p -> {
            if (p != sender) p.notify(sender, message);
        });
    }
}
```

Die geänderte Klasse Participant:

```
package appl;

public class Participant {

    private final Chatroom chatroom;
    private final String name;

    public Participant(Chatroom chatroom, String name) {
        this.chatroom = chatroom;
        this.chatroom.add(this);
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void send(String message) {
        this.chatroom.send(this, message);
    }

    public void notify(Participant sender, String message) {
        System.out.printf(
            "User %s received message: %s (Sender = %s)\n",
            this.name, message, sender.name);
    }
}
```

Und hier die wesentlich einfachere Test-Applikation:

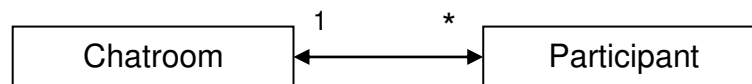
```
package appl;

public class Application {

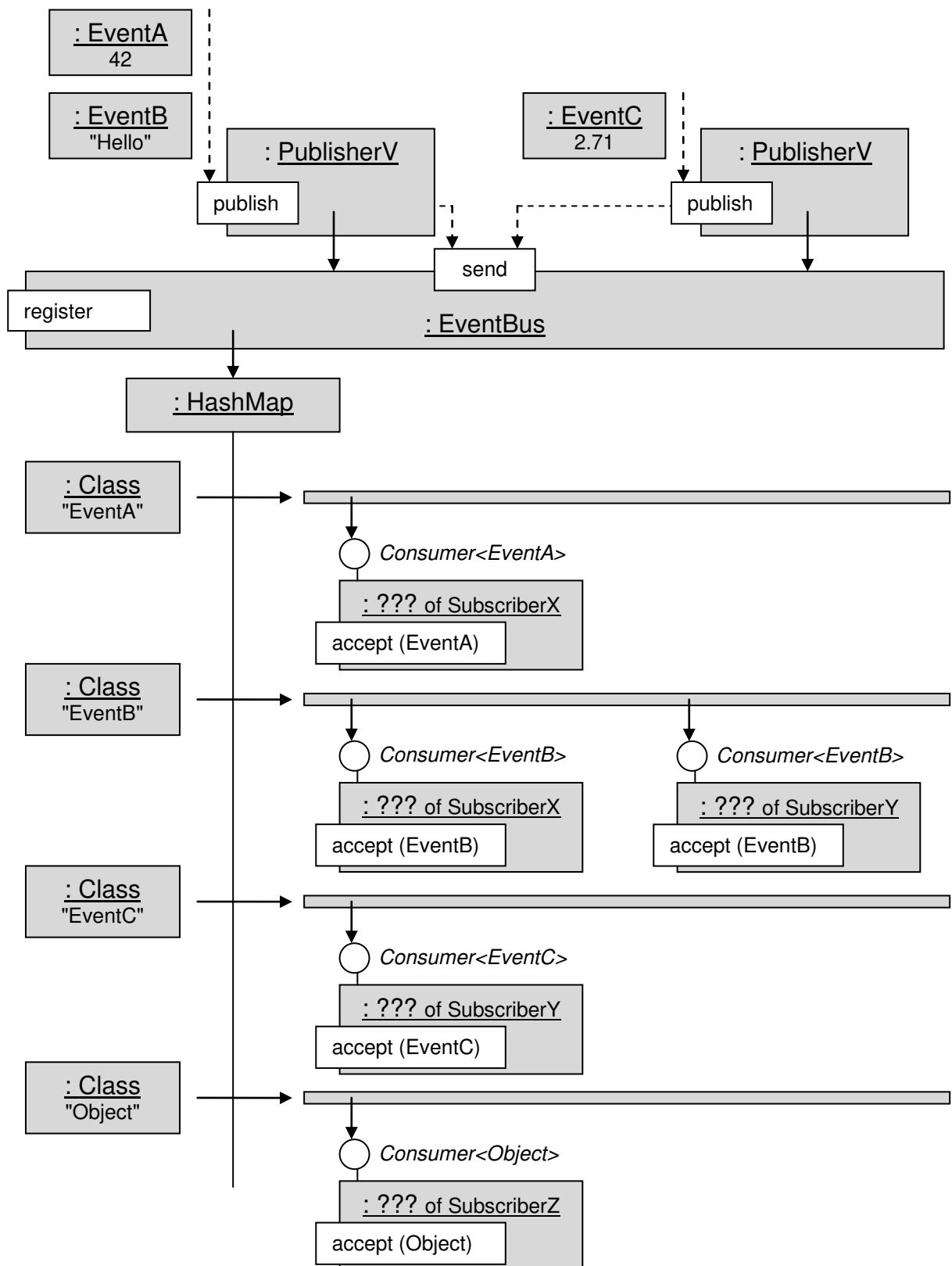
    public static void main(String[] args) {
        final Chatroom chatroom = new Chatroom();
        final Participant meier = new Participant(chatroom, "Meier");
        final Participant mueller = new Participant(chatroom,
"Mueller");
        final Participant franke = new Participant(chatroom,
"Franke");
        meier.send("Glueck");
        mueller.send("Glanz");
        franke.send("Ruhm");
    }
}
```

Die Ausgaben sind dieselben wie im letzten Abschnitt.

Das Klassendiagramm:



4.9.3 EventBus



Eine Publish-Subscriber-Anwendung kann als "verfeinerte" Form des Chatrooms angesehen werden.

Publisher veröffentlichen Events (oder Nachrichten); Subscriber interessieren sich für Events (für Nachrichten). Publisher können Events verschiedenen Typs veröffentlichen: Sport-Events, Politik-Events, Yellow-Events etc. Subscriber interessieren sich jeweils nur für spezifische Events – der eine Subscriber nur für Sport, der andere nur für Politik und der dritte für alles Mögliche.

In Sinne der losen Kopplung sollten weder die Publisher die Subscriber kennen noch umgekehrt die Subscriber die Publisher. Sie werden über einen Mediator zusammengeführt – über einen Event-Bus.

Hier zunächst einige Event-Typen (Sport, Politik, Yellow):

```
package appl;

public class EventA {
    public final int a;
    public EventA(int a) {
        this.a = a;
    }
    @Override
    public String toString() {
        return this.getClass().getSimpleName() + " [" + this.a + "]";
    }
}
```

```
package appl;

public class EventB {
    public final String b;
    public EventB(String b) {
        this.b = b;
    }
    @Override
    public String toString() {
        return this.getClass().getSimpleName() + " [" + this.b + "]";
    }
}
```

```
package appl;

public class EventC {
    public final double c;
    public EventC(double c) {
        this.c = c;
    }
    @Override
    public String toString() {
        return this.getClass().getSimpleName() + " [" + this.c + "]";
    }
}
```

Man beachte, dass die Event-Klassen von keinerlei Basisklasse abgeleitet sind!

Es gibt zwei Publisher, die unterschiedliche Events feuern wollen. Beiden Publishern wird ein `EventBus` übergeben, auf welchen sie dessen `send`-Methode aufrufen (der `EventBus` wird weiter unten vorgestellt). Der erste feuert einen `EventA` und einen `EventB`; der zweite eine `EventC`:

```
package appl;

import util.EventBus;

public class PublisherV {
    private final EventBus eventBus;

    public PublisherV(EventBus eventBus) {
        this.eventBus = eventBus;
    }

    public void pushlish() {
        this.eventBus.send(new EventA(42));
        this.eventBus.send(new EventB("Hello"));
    }
}
```

```
package appl;

import util.EventBus;

public class PublisherW {
    private final EventBus eventBus;

    public PublisherW(EventBus eventBus) {
        this.eventBus = eventBus;
    }

    public void pushlish() {
        this.eventBus.send(new EventC(2.71));
    }
}
```

Die `send`-Methode des `EventBus` ist sehr "anspruchslos": sie ist mit jedem Object zufrieden.

Es existieren drei Subscriber: `SubscriberX`, `SubscriberY` und `SubscriberZ`.

Der erste Subscriber interessiert sich für `EventA`- und `EventB`-Objekte:

```
package appl;

import java.util.function.Consumer;
import util.EventBus;

public class SubscriberX {

    public SubscriberX(EventBus eventBus) {

        eventBus.register(EventA.class, new Consumer<EventA>() {
            @Override
            public void accept(EventA e) {
                System.out.println("\t\t" +
                    this.getClass().getName() + " <- " + e.a);
            }
        });

        eventBus.register(EventB.class, new Consumer<EventB>() {
            @Override
            public void accept(EventB e) {
                System.out.println("\t\t" +
                    this.getClass().getName() + " <- " + e.b);
            }
        });
    }
}
```

Der Konstruktor ruft zweimal die `register`-Methode auf den `EventBus` auf. Er übergibt dabei jeweils den Typ des Events und einen `Consumer`, dessen `accept`-Methode mit einem Event des entsprechenden Typs aufgerufen werden kann. Man beachte, dass die ganze Konstruktion dank Generics typsicher ist.

Der zweite Subscriber interessiert sich für Events des Typs `EventB` und `EventC`. Statt anonymer Klassen werden die `Consumer` nun aber weniger geschwätzig mittels Lambdas implementiert:

```
package appl;

import util.EventBus;

public class SubscriberY {
    public SubscriberY(EventBus eventBus) {

        eventBus.register(EventB.class,
            e -> System.out.println("\t\t" +
                this.getClass().getName() + " <- " + e.b));

        eventBus.register(EventC.class,
            e -> System.out.println("\t\t" +
                this.getClass().getName() + " <- " + e.c));
    }
}
```

Der dritte Subscriber interessiert sich für alles Mögliche:

```
package appl;

import util.EventBus;

public class SubscriberZ {
    public SubscriberZ(EventBus eventBus) {
        eventBus.register(Object.class,
            e -> System.out.println("\t\t" +
                this.getClass().getName() + " <- " + e));
    }
}
```

Die Demo-Applikation:

```
package appl;

import util.EventBus;

public class Application {

    public static void main(String[] args) {
        final EventBus bus = new EventBus();

        new SubscriberX(bus);
        new SubscriberY(bus);
        new SubscriberZ(bus);

        final PublisherV v = new PublisherV(bus);
        final PublisherW w = new PublisherW(bus);

        v.pushlish();
        w.pushlish();
    }
}
```

Und dessen Ausgaben:

```
> EventA [42]
    appl.SubscriberX$1 <- 42
    appl.SubscriberZ <- EventA [42]
> EventB [Hello]
    appl.SubscriberX$2 <- Hello
    appl.SubscriberY <- Hello
    appl.SubscriberZ <- EventB [Hello]
> EventC [2.71]
    appl.SubscriberY <- 2.71
    appl.SubscriberZ <- EventC [2.71]
```

nd hier schließlich der wichtigste Teil des Ganzen: die `util`-Klasse `EventBus`:

```
package util;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Consumer;

public class EventBus {

    private final Map<Class<?>, List<Consumer<?>>> map =
        new HashMap<>();

    public synchronized <T> void register(
        Class<T> type, Consumer<T> consumer) {
        List<Consumer<?>> consumers = this.map.get(type);
        if (consumers == null) {
            consumers = new ArrayList<Consumer<?>>();
            this.map.put(type, consumers);
        }
        consumers.add(consumer);
    }

    @SuppressWarnings("unchecked")
    public void send(Object event) {
        System.out.println("> " + event);
        for(Class<?> eventClass = event.getClass();
            eventClass != null;
            eventClass = eventClass.getSuperclass()) {
            final List<Consumer<?>> consumers =
this.map.get(eventClass);
            if (consumers != null) {
                for (final Consumer<?> consumer : consumers) {
                    ((Consumer<Object>) consumer).accept(event);
                }
            }
        }
    }
}
```

Das genaue Studium sei dem Leser / der Leserin überlassen...

4.9.4 Queue

Multithreaded-Anwendungen basieren häufig auf der Beziehung `Producer` – `Consumer`. Ein `Producer` produziert in einem eigenen Thread Produkte, welche an einen `Consumer` weitergereicht werden müssen, welcher ebenfalls in einem eigenen Thread läuft und diese Produkte dann verarbeiten muss. Statt eines einzigen können auch viele `Producer` existieren, welche alle in separaten Threads laufen. Dasselbe gilt für die `Consumer`.

`Producer` und `Consumer` müssen entkoppelt werden. I.d.R. verwendet man zum Zwecke dieser Entkopplung eine `Queue`:

```
package util;

import java.util.LinkedList;

public class Queue<T> {

    private final LinkedList<T> list = new LinkedList<>();

    private final int max = 3;

    public void enqueue(T product) {
        this.log(">> enqueue");
        synchronized (this) {
            while (this.list.size() == this.max) {
                this.log("WAITING...");
                try {
                    this.wait();
                }
                catch (final InterruptedException ignored) {}
            }
            assert this.list.size() < this.max;
            this.list.add(product);
            this.notifyAll();
        }
        this.log("<< enqueue " + product);
    }

    public T dequeue() {
        this.log("\t\t>> dequeue");
        T product = null;
        synchronized (this) {
            while (this.list.size() == 0) {
                this.log("\t\tWAITING...");
                try {
                    this.wait();
                }
                catch (final InterruptedException ignored) {}
            }
            assert this.list.size() > 0;
            product = this.list.removeFirst();
            this.notifyAll();
        }
    }
}
```

```

        this.log("\t\t<< dequeue " + product);
        return product;
    }

    private void log(String text) {
        System.out.println(text + " [" +
            Thread.currentThread().getId() + "]");
    }
}

```

Objekte der obigen `Queue`-Klasse besitzen eine `List<T>`, in der max viele Elemente abgestellt werden können. `enqueue` wird vom `Producer` aufgerufen werden, um ein neues Produkt (vom Typ `T`) in der `Queue` abzustellen. Die Methode muss eventuell warten, bis die `Queue` nicht mehr voll ist. `dequeue` wird vom `Consumer` aufgerufen werden, um aus der `Queue` das älteste Produkt herauszuholen. `dequeue` muss ggf. warten, bis die `Queue` nicht mehr leer ist.

Hier die Klassen `Producer` und `Consumer`:

```

package appl;

import util.Queue;
import util.Util;

public class Producer extends Thread {

    private final Queue<String> queue;

    public Producer(Queue<String> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            final int sleeptime = (int) (Math.random() * 1000);
            Util.sleep(sleeptime);
            this.queue.enqueue "\"" + i + "\"");
        }
    }
}

```

```

package appl;

import util.Queue;
import util.Util;

public class Consumer extends Thread {

    private final Queue<String> queue;

    public Consumer(Queue<String> queue) {
        this.queue = queue;
    }
}

```

```
@Override
public void run() {
    final int sleeptime = 1000;
    while (true) {
        final String product = this.queue.dequeue();
        if (product.length() == 0)
            break;
        Util.sleep(sleeptime);
    }
}
```

Sowohl `Producer` als auch `Consumer` müssen die `Queue` kennen, welche sie als Vermittler benutzen. Die `run`-Methoden beider Klassen werden in separaten Threads aufgerufen werden. Der `Producer` produziert fünf Produkte und terminiert dann. Der `Consumer` terminiert dann, wenn er einen leeren String erhält. Sowohl `Producer` als auch `Consumer` brauchen jeweils eine bestimmte Zeit (die vom `Producer` per `Math.random` berechnet wird), um die Produkte zu erstellen resp. zu verbrauchen.

In der folgenden beispielhaften Applikation verbrauchen zwei `Consumer` die Produkte, die von einem einzigen `Producer` erstellt werden:

```
package appl;

import util.Queue;
import util.Util;

public class Application {

    public static void main(String[] args) {

        final Queue<String> queue = new Queue<>();

        final Consumer consumer1 = new Consumer(queue);
        final Consumer consumer2 = new Consumer(queue);
        final Producer producer = new Producer(queue);

        consumer1.start();
        Util.sleep(1000);
        consumer2.start();
        Util.sleep(1000);
        producer.start();

        try {
            producer.join();
        }
        catch (final Exception e) {
            throw new RuntimeException(e);
        }

        queue.enqueue("");
        queue.enqueue("");
    }
}
```

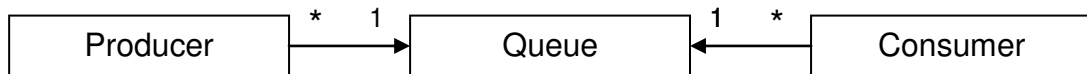

Hier eine mögliche Ausgabe (die Ausgaben des `Producers` finden sich in der linken Spalte; die Ausgaben der beiden `Consumer` in der mittleren resp. rechten Spalte):

```

>> dequeue [10]
WAITING... [10]
>> dequeue [11]
WAITING... [11]
>> enqueue [12]
<< enqueue "0" [12]
    << dequeue "0" [11]
        WAITING... [10]
>> enqueue [12]
    << dequeue "1" [10]
<< enqueue "1" [12]
    >> dequeue [11]
        WAITING... [11]
>> enqueue [12]
    << dequeue "2" [11]
<< enqueue "2" [12]
    >> dequeue [10]
        WAITING... [10]
>> enqueue [12]
<< enqueue "3" [12]
    << dequeue "3" [10]
    >> dequeue [11]
        WAITING... [11]
>> enqueue [12]
    << dequeue "4" [11]
<< enqueue "4" [12]
>> enqueue [1]
<< enqueue [1]
>> enqueue [1]
<< enqueue [1]
    >> dequeue [10]
    << dequeue [10]
    >> dequeue [11]
    << dequeue [11]

```

Das Klassendiagramm:



Aufgaben

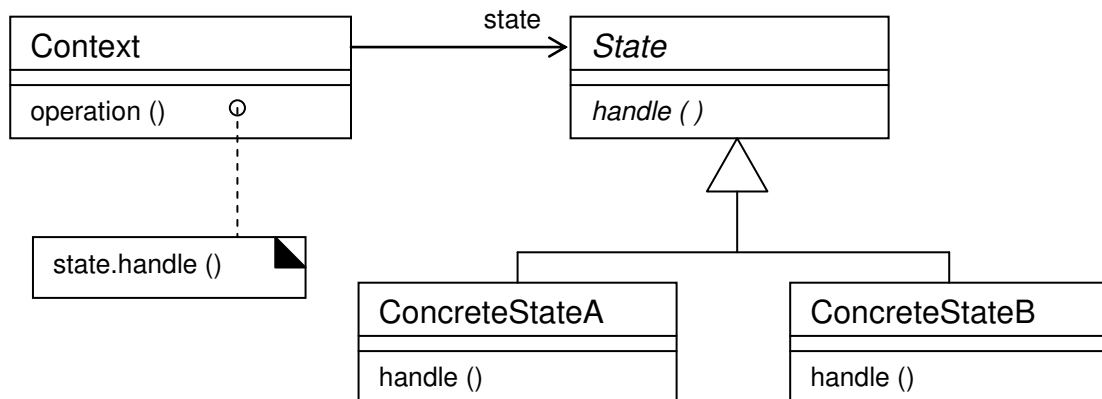
Parametrisieren Sie die Anwendung derart, dass die `Consumer` sehr langsam sind – mit der Folge, dass der `Producer` sehr häufig warten muss. Und andersherum: Lassen Sie die `Consumer` schnell konsumieren – damit sie häufig auf ein neues Produkt des `Producers` warten müssen. Studieren Sie die Ausgaben!

Wie verhält sich das System, wenn die Anzahl der Listeneinträge in der `Queue` nicht begrenzt wird?

Wie verhält sich das System, wenn die maximale Anzahl der `Queue`-Einträge sehr klein ist?

4.10 State

"Ermögliche es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hätte." (Gamma, 397)



4.10.1 Problem

In einem Programmsystem zur Zugangskontrolle existieren Türen. Eine Tür kann geöffnet oder geschlossen sein. Und demgemäß kann man eine Tür schließen und öffnen. Aber man kann sie nur im geschlossenen Zustand öffnen, und nur im geöffneten Zustand schließen.

Hier zunächst eine einfache Lösung des Problems:

```
package appl;

public class Door {

    private boolean isOpen;

    public void open() {
        if (this.isOpen)
            throw new IllegalStateException("door is open");
        System.out.println("closed --> opened");
        this.isOpen = true;
    }

    public void close() {
        if (! this.isOpen)
            throw new IllegalStateException("door is closed");
        System.out.println("opened --> closed");
        this.isOpen = false;
    }

    public boolean isOpen() {
        return this.isOpen;
    }
}
```

Eine Test-Anwendung:

```
package appl;

public class Application {
    public static void main(String[] args) {

        final Door door = new Door();
        try {
            door.open();
            door.close();
            door.open();
            door.open();
        }
        catch (final IllegalStateException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Die Ausgaben:

```
closed --> opened
opened --> closed
closed --> opened
door is open
```

Die Lösung ist klar und übersichtlich. Man stelle sich nun aber einen Kontext vor, welcher viele mögliche Zustände hat – nicht nur zwei. Dementsprechend viele Methoden wird es geben, die den jeweiligen Zustand ändern. Das Verhalten all dieser Methoden wird natürlich i.d.R. vom aktuellen Zustand abhängen. All diese Methoden würden also einen umfangreichen "switch" beinhalten müssen. Und wären damit nicht sehr übersichtlich und auch nicht änderungsfreundlich.

Besser ist es, die unterschiedlichen Zustände eines Systems durch Objekte entsprechender Zustandsklassen zu repräsentieren. Diese Zustandsklassen werden dann natürlich von einer gemeinsamen Basis-Klasse abgeleitet werden – einer Basisklasse, in welcher bereits die Referenz auf den Kontext implementiert ist.

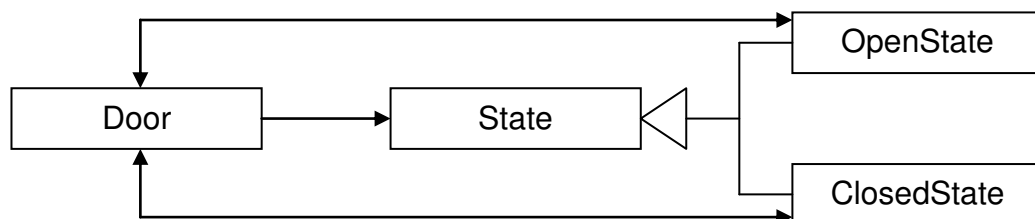
Im Falle der obigen `Door`-Klasse schießt man dann natürlich mit Kanonen auf Spatzen – aber warum einfach, wenn's auch kompliziert geht?

4.10.2 Lösung

Für jeden Zustand wird eine eigene Klasse definiert. Diese Zustandsklassen sind von einer gemeinsamen Basisklasse abgeleitet. Die Basisklasse spezifiziert für jede zustandsrelevante Methode der Kontext-Klasse (`Door`) eine gleichnamige Methode. Die zustandsrelevanten Methoden des Kontextes delegieren dann einfach an die entsprechende Methode desjenigen Zustands-Objekts, welches den aktuellen Zustand des Systems repräsentiert.

Der Kontext referenziert alle Zustands-Objekte. Er besitzt eine weitere Referenz auf das aktuelle Zustands-Objekt. Jedes Zustandsobjekt besitzt umgekehrt eine Referenz auf den Kontext (`Door`).

Das Klassendiagramm:



Die Basisklasse `State`:

```
package appl;

public abstract class State {

    protected final Door door;

    public State(Door door) {
        this.door = door;
    }

    public void open() {
        throw new IllegalStateException(this.door.getState()
            + " cannot open");
    }

    public void close() {
        throw new IllegalStateException(this.door.getState()
            + " cannot close");
    }

    public String toString() {
        return this.getClass().getName();
    }
}
```

Die Klasse OpenState:

```
package appl;

public class OpenedState extends State {

    public OpenedState(Door door) {
        super(door);
    }

    @Override
    public void close() {
        this.door.setState(this.door.closedState);
    }
}
```

Die Klasse ClosedState:

```
package appl;

public class ClosedState extends State {

    public ClosedState(Door door) {
        super(door);
    }

    @Override
    public void open() {
        this.door.setState(this.door.openedState);
    }
}
```

Die Klasse Door:

```
package appl;

public class Door {

    final OpenedState openedState = new OpenedState(this);
    final ClosedState closedState = new ClosedState(this);

    private State currentState = this.closedState;

    public State getState() {
        return this.currentState;
    }

    void setState(State state) {
        System.out.println(this.currentState + " --> " + state);
        this.currentState = state;
    }

    public void open() {
        this.currentState.open();
    }

    public void close() {
        this.currentState.close();
    }
}
```

Alle zustandsändernden Methoden (`open` und `close`) delegieren nun einfach an eine jeweils namesgleiche Methode einer der `State`-Klassen.

Als Test-Anwendung kann dasselbe Programm benutzt werden wie im letzten Abschnitt.

Aufgaben

Warum werfen die `open`- und die `close`-Methode der `State`-Klasse beide Exceptions?

An welchen Stellen müssen die bestehenden Klassen erweitert werden, wenn ein weiterer Status hinzukommt ("Halb-Offen")?

Wo könnte man zusätzliche Instanzvariablen definieren, die statusspezifisch sind? Wo könnten Variablen definiert werden, welche statusübergreifend sind?

4.10.3 Stack

Ein Stack mit begrenzter Kapazität kann drei Zustände annehmen: leer, gefüllt und voll. Hier eine State-Pattern basierte Implementierung eines solchen Stacks:

```
package appl;

public class Stack<T> {

    private abstract class State {
        public final String displayName;

        public State(String displayName) {
            this.displayName = displayName;
        }

        public void push(T element) { throw new ... }
        public T pop() { throw new IllegalStateException("..."); }
        public T top() { throw new IllegalStateException("..."); }

        public boolean isEmpty() { return false; }
        public boolean isFilled() {return false; }
        public boolean isFull() { return false; }
    }

    private class EmptyState extends State {
        public EmptyState() { super("empty"); }
        @Override
        public void push(T element) {
            Stack.this.elements[Stack.this.count++] = element;
            Stack.this.setCurrentState(Stack.this.filledState);
        }
        @Override
        public boolean isEmpty() { return true; }
    }

    private class FilledState extends State {
        public FilledState() { super("filled"); }
        @Override
        public void push(T element) {
            Stack.this.elements[Stack.this.count++] = element;
            if (Stack.this.count == Stack.this.elements.length)
                Stack.this.setCurrentState(Stack.this.fullState);
            else
                Stack.this.setCurrentState(Stack.this.filledState);
        }
        @Override
        public T pop() {
            Stack.this.count--;
            if (Stack.this.count == 0)
                Stack.this.setCurrentState(Stack.this.emptyState);
            else
                Stack.this.setCurrentState(Stack.this.filledState);
            return Stack.this.elements[Stack.this.count];
        }
        @Override
        public T top() {
            return Stack.this.elements[Stack.this.count - 1];
        }
    }
}
```

```

    }
    @Override
    public boolean isFilled() { return true; }

}

private class FullState extends State {
    public FullState() { super("full"); }
    @Override
    public T pop() {
        Stack.this.count--;
        Stack.this.setCurrentState(Stack.this.filledState);
        return Stack.this.elements[Stack.this.count];
    }
    @Override
    public T top() {
        return Stack.this.elements[Stack.this.count - 1];
    }
    @Override
    public boolean isFull() { return true; }
}

private final T[] elements;
private int count;

private final State emptyState = new EmptyState();
private final State filledState = new FilledState();
private final State fullState = new FullState();

private State currentState;

@SuppressWarnings("unchecked")
public Stack(int size) {
    if (size < 2)
        throw new IllegalArgumentException("size must be >= 2");
    this.elements = (T[]) new Object[size];
    this.currentState = this.emptyState;
}

private void setCurrentState(State state) {
    System.out.printf("%s ==> %s\n",
        this.currentState.displayName, state.displayName);
    this.currentState = state;
}

public void push(T element) { this.currentState.push(element); }
public T pop() { return this.currentState.pop(); }
public T top() { return this.currentState.top(); }
public boolean isEmpty() { return this.currentState.isEmpty(); }
public boolean isFull() { return this.currentState.isFull(); }
public boolean isFilled() { return this.currentState.isFilled(); }
}

```

Hier eine Test-Anwendung:

```
package appl;

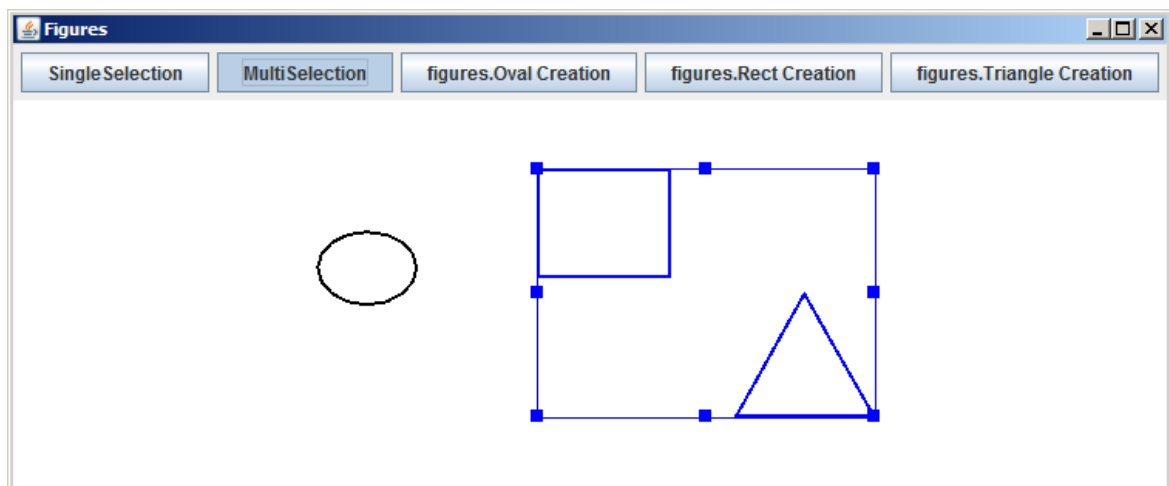
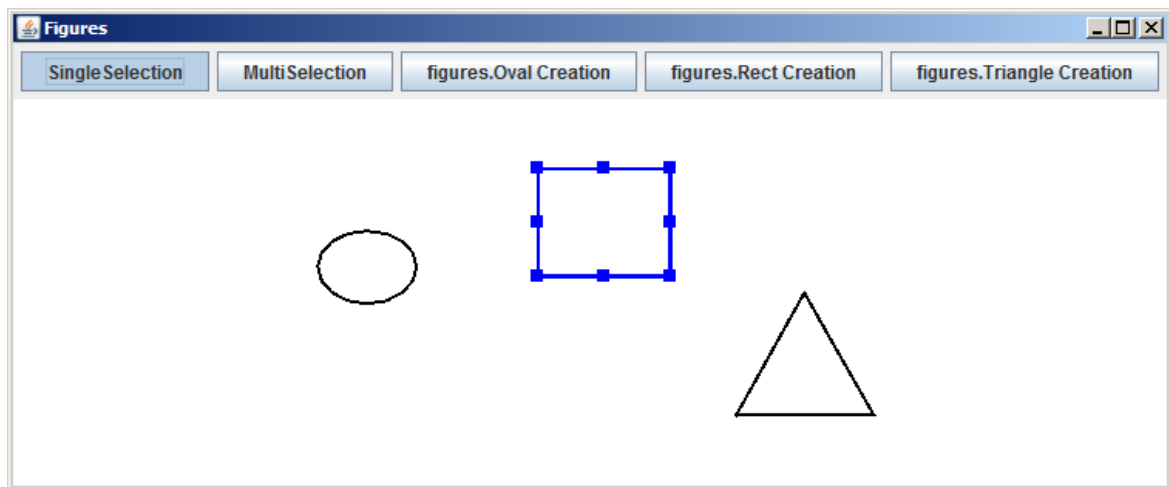
public class Application {
    public static void main(String[] args) {
        final Stack<Integer> stack = new Stack<>(5);
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.push(40);
        stack.push(50);
        try {
            stack.push(60);
        }
        catch (final Exception e) {
            System.out.println(e);
        }
        while (!stack.isEmpty())
            System.out.println(stack.pop());
        try {
            stack.pop();
        }
        catch (final Exception e) {
            System.out.println(e);
        }
    }
}
```

Die Ausgaben:

```
empty ==> filled
filled ==> filled
filled ==> filled
filled ==> filled
filled ==> full
java.lang.IllegalStateException: Push: error (current state = full)
full ==> filled
50
filled ==> filled
40
filled ==> filled
30
filled ==> filled
20
filled ==> empty
10
java.lang.IllegalStateException: Pop: error (current state = empty)
```

4.10.4 FigureEditor

Das folgende Beispiel sei dem Leser zum Selbststudium überlassen.



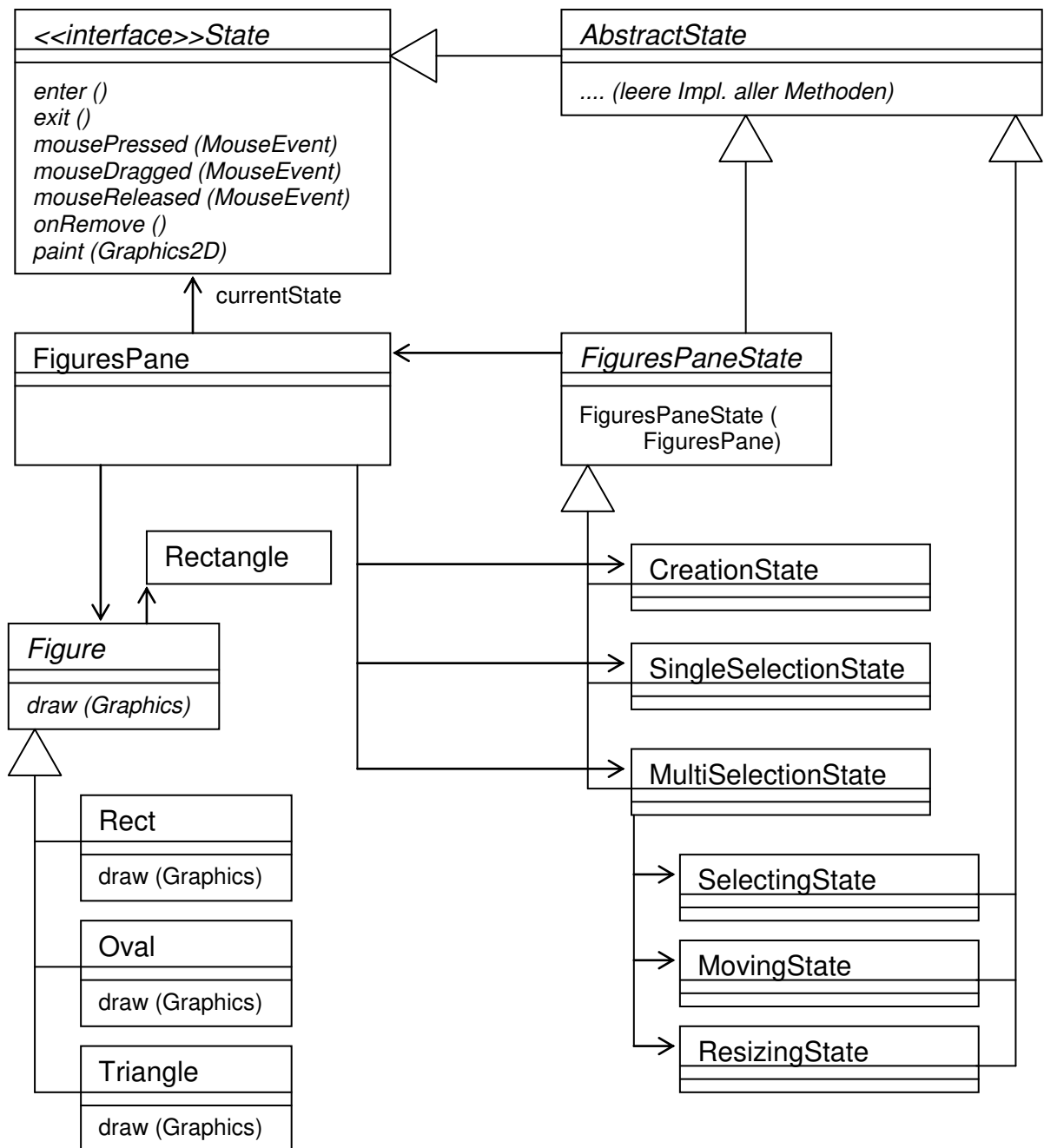
Hier die zentrale Basisklasse State:

```
package core;

import java.awt.Graphics2D;
import java.awt.event.MouseEvent;

public interface State {
    public abstract void enter();
    public abstract void exit();
    public abstract void mousePressed(MouseEvent e);
    public abstract void mouseReleased(MouseEvent e);
    public abstract void mouseDragged(MouseEvent e);
    public abstract void onRemove();
    public abstract void paint(Graphics2D g);
}
```

Ein (etwas vereinfachtes) Klassendiagramm:



4.10.5 Dispatcher

Die folgende Anwendung verfolgt einen etwas anderen Ansatz als die bisherigen Beispiele. Die Kontext-Objekte der letzten Beispiele besitzen individuelle Methoden (z.B. `Open`, `Close`), um "Ereignisse" an das System heranzutragen. Alternativ dazu könnte man eine einzige Operation vorsehen, welcher ein Event-Objekt übergeben wird. Event-Objekte sind Objekte von Klassen, die von einem allgemeinen `Event`-Interface abgeleitet sind.

Die Event-Klassen sind abgeleitet von dem Interface `Event` (welches als reines Marker-Interfaces fungiert):

```
package util;

public interface Event { }
```

Hier zwei beispielhafte Event-Klassen:

```
package appl;

import util.Event;

public class EventA implements Event {
    public final int a;

    public EventA(int a) {
        this.a = a;
    }
}
```

```
package appl;

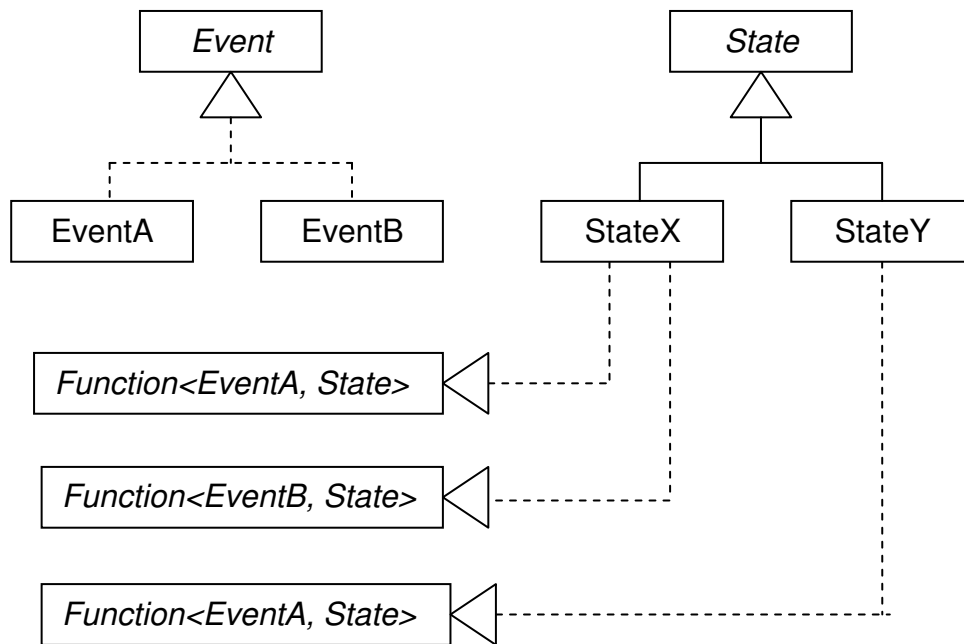
import util.Event;

public class EventB implements Event {
    public final String b;

    public EventB(String b) {
        this.b = b;
    }
}
```

Die Status-Klassen sind abgeleitet vom der Basisklasse `State`. Konkrete, von `State` abgeleitete Klassen registrieren für jeden `Event`-Typ, den sie sinnvoll verarbeiten können, ein Objekt, dessen Klasse das Interfaces `Function` implementiert. Die `Function` bildet einen `Event` auf einen `State` ab (auf einen `State`, der nach Ausführung der `Function` als neuer aktueller Zustand fungiert). In der `Function` ist die eigentliche jeweils spezifische Verarbeitung implementiert (wobei die Verarbeitung auch die Initiierung von Transitionen umfasst).

Das Klassendiagramm:



Die Basisklasse State:

```

package util;

import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

public abstract class State {

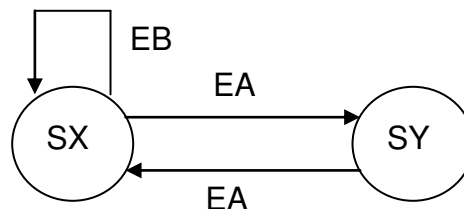
    public final String name;
    public State(String name) {
        this.name = name;
    }
    private final Map<Class<? extends Event>,
        Function<? extends Event, State>> functions = new HashMap<>();

    public <T extends Event> void register(
        Class<T> eventType, Function<T, State> function) {
        this.functions.put(eventType, function);
    }

    @SuppressWarnings({ "unchecked", "rawtypes" })
    public State dispatch(Event e) {
        final Function<? extends Event, State> function =
            this.functions.get(e.getClass());
        if (function == null) {
            System.out.printf(
                "Cannot dispatch. Current State = %s, Event = %s\n",
                this.name, e.getClass().getSimpleName());
            return this;
        }
        final State newState = (State)((Function)function).apply(e);
        System.out.printf("%s ==> %s\n", this.name, newState.name);
        return newState;
    }
}

```

Im zu implementierenden Kontext sind folgende Statusübergänge vorgesehen:




```
package appl;

import java.util.function.Function;
import util.Event;
import util.State;

public class MyContext {

    private class StateX extends State {
        public StateX() {
            super("StateX");
            // Impl. mit anonymer Klasse
            this.register(EventA.class, new Function<EventA, State>() {
                @Override
                public State apply(EventA e) {
                    System.out.println("\t" + StateX.this.name +
                        " : Processing EventA : " + e.a);
                    return MyContext.this.stateY;
                }
            });
            // Impl. mit Lambda
            this.register(EventB.class, e -> {
                System.out.println("\t" + this.name +
                    " : Processing EventB : " + e.b);
                return MyContext.this.stateX;
            });
        }
    }

    private class StateY extends State {
        public StateY() {
            super("StateY");
            this.register(EventA.class, e -> {
                System.out.println("\t" + this.name +
                    " : Processing EventA : " + e.a);
                return MyContext.this.stateX;
            });
        }
    }

    private final StateX stateX = new StateX();
    private final StateY stateY = new StateY();

    private State currentState = this.stateX;

    public void dispatch(Event e) {
        this.currentState = this.currentState.dispatch(e);
    }
}
```

Die Test-Anwendung:

```
package appl;

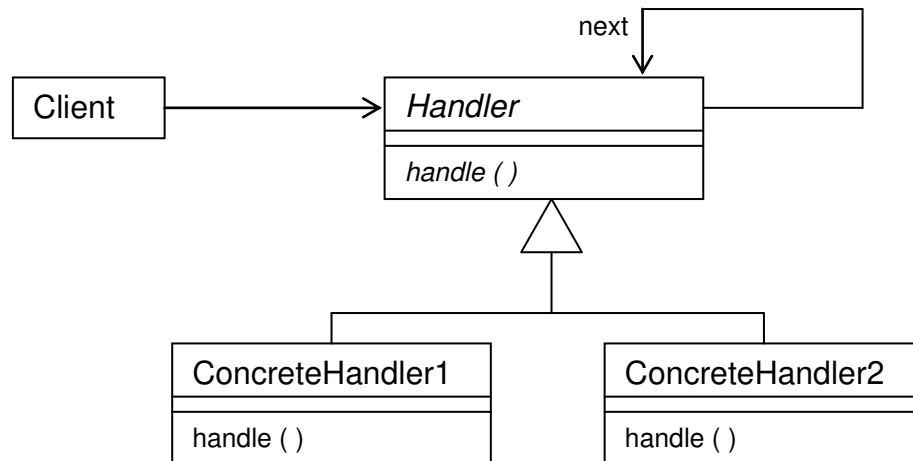
public class Application {
    public static void main(String[] args) {
        final MyContext context = new MyContext();
        context.dispatch(new EventA(4711));
        context.dispatch(new EventA(4712));
        context.dispatch(new EventB("Hello"));
        context.dispatch(new EventA(4713));
        try {
            context.dispatch(new EventB("World"));
        }
        catch (final Exception e) {
            System.out.println(e);
        }
        context.dispatch(new EventA(4714));
    }
}
```

Die Ausgaben:

```
StateX : Processing EventA : 4711
StateX ==> StateY
StateY: Processing EventA : 4712
StateY ==> StateX
StateX : Processing EventB : Hello
StateX ==> StateX
StateX : Processing EventA : 4713
StateX ==> StateY
Cannot dispatch. Current State = StateY, Event = EventB
StateY: Processing EventA : 4714
StateY ==> StateX
```

4.11 Chain of Responsibility

"Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Aufgabe zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt." (Gamma, 367)



4.11.1 Problem

Die folgende Anwendung kann "Probleme" lösen. Die Probleme werden vom Benutzer eingegeben (Ausgaben sind fett):

```
22 + 33
55
sqr 4
16
concat hello world
helloworld
33 - 22
cannot solve Problem
exit
```

Hier eine einfache Anwendung, welche Probleme der oben skizzierten Art lösen kann:

```
package appl;
// ...
public class Application {
    public static void main(String[] args) throws Exception {
        System.out.println("a problem (or exit)");
        final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        for (String problem = reader.readLine();
            ! "exit".equals(problem);
            problem = reader.readLine()) {
            final String[] tokens = problem.split(" ");
            if (!solve(tokens))
                System.out.println("Problem cannot be solved");
        }
    }

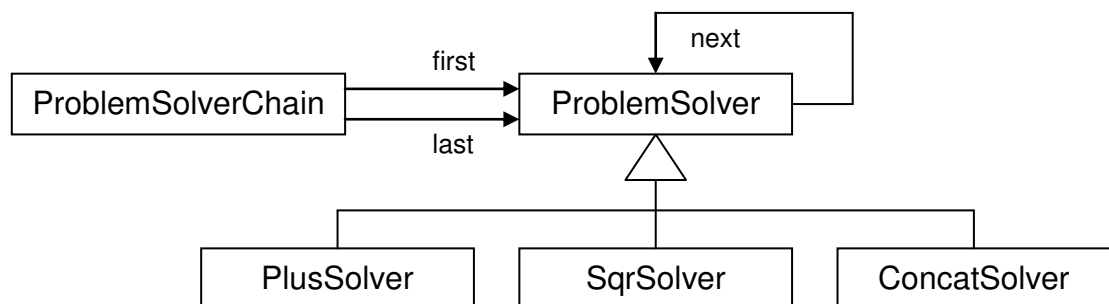
    static private boolean solve(String[] tokens) {
        if (tokens.length == 3 && tokens[1].equals("+")) {
            final int x = Integer.parseInt(tokens[0]);
            final int y = Integer.parseInt(tokens[2]);
            final int sum = x + y;
            System.out.println(sum);
            return true;
        }
        if (tokens.length == 2 && tokens[0].equals("sqr")) {
            final int v = Integer.parseInt(tokens[1]);
            final int sqr = v * v;
            System.out.println(sqr);
            return true;
        }
        if (tokens.length == 3 && tokens[0].equals("concat")) {
            System.out.println(tokens[1] + tokens[2]);
            return true;
        }
        return false;
    }
}
```

Immer dann, wenn eine neue Sorte von Problemen hinzukommt, muss der "switch" der `solve`-Methode erweitert werden. Es wäre schön, wenn man den switch vermeiden könnte.

4.11.2 Lösung

Für jede Sorte von Problemen wird eine eigene Klasse erstellt, welche von einer abstrakten Basisklasse (`ProblemSolver`) abgeleitet wird und genau einmal instanziiert wird. Die Instanzen dieser Klassen werden miteinander verkettet. Das Problem, welches der Benutzer eingegeben hat, wird dann an der Kette dieser Problemlöser entlanggeführt. Kann einer dieser Problemlöser das Problem lösen (ist er für das Problem zuständig), so löst er das Problem. Ansonsten wird das Problem einfach dem nächsten in der Kette enthaltenen Problemlöser weitergereicht.

Das Klassendiagramm:



Die Basisklasse `ProblemSolver`:

```

package util;

public abstract class ProblemSolver {
    private ProblemSolver next;

    public abstract boolean solve(String[] tokens);

    public void setNext(ProblemSolver next) {
        this.next = next;
    }

    protected boolean sendToNext(String[] tokens) {
        if (this.next != null)
            return this.next.solve(tokens);
        return false;
    }
}
  
```

`solve` muss von jeder abgeleiteten Klasse implementiert werden. Die Methode muss `true` liefern, sofern das Problem gelöst wurde. Ansonsten kann die Methode `sendToNext` aufrufen, um das Problem an den Nachfolger weiterzureichen.

Die Klasse `ProblemSolverChain`:

```
package util;

public class ProblemSolverChain {

    private ProblemSolver first;
    private ProblemSolver last;

    public void append(ProblemSolver problemSolver) {
        if (this.first == null)
            this.first = problemSolver;
        else
            this.last.setNext(problemSolver);
        this.last = problemSolver;
    }

    public boolean solve(String[] tokens) {
        return this.first != null && this.first.solve(tokens);
    }
}
```

Ein `ProblemSolverChain`-Objekt dient als Registratur, bei welcher `ProblemSolver` registriert werden können (`append`). Die `solve`-Methode reicht das ihr übergebene Problem an den ersten `ProblemSolver` der Kette weiter.

Hier einige konkrete `ProblemSolver`:

```
package appl;

import util.ProblemSolver;

public class PlusSolver extends ProblemSolver {
    @Override
    public boolean solve(String[] tokens) {
        if (!(tokens.length == 3 && tokens[1].equals("+")))
            return this.sendToNext(tokens);
        final int x = Integer.parseInt(tokens[0]);
        final int y = Integer.parseInt(tokens[2]);
        final int result = x + y;
        System.out.println(result);
        return true;
    }
}
```

```
package appl;

import util.ProblemSolver;

public class SqrSolver extends ProblemSolver {
    @Override
    public boolean solve(String[] tokens) {
        if (!(tokens.length == 2 && tokens[0].equals("sqr")))
            return this.sendToNext(tokens);
        final int value = Integer.parseInt(tokens[1]);
        final int result = value * value;
        System.out.println(result);
        return true;
    }
}
```

```
package appl;

import util.ProblemSolver;

public class ConcatSolver extends ProblemSolver {
    @Override
    public boolean solve(String[] tokens) {
        if (!(tokens.length == 3 && tokens[0].equals("concat")))
            return this.sendToNext(tokens);
        final String result = tokens[1] + tokens[2];
        System.out.println(result);
        return true;
    }
}
```

Und hier schließlich die Test-Anwendung:

```
package appl;
// ...
import util.ProblemSolverChain;

public class Application {
    public static void main(String[] args) throws Exception {

        final ProblemSolverChain chain = new ProblemSolverChain();

        chain.append(new PlusSolver());
        chain.append(new SqrSolver());
        chain.append(new ConcatSolver());

        System.out.println("a problem (or exit)");
        final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        for (String problem = reader.readLine();
            ! "exit".equals(problem);
            problem = reader.readLine()) {
            final String[] tokens = problem.split(" ");
            if (! chain.solve(tokens))
                System.out.println("Problem cannot be solved");
        }
    }
}
```

Die Erzeugung und Registrierung der `ProblemSolver` könnte natürlich auch "generisch" erfolgen. Die Namen der zu instanziiierenden `ProblemSolver`-Klassen könnten in einer Textdatei hinterlegt sein. Per `Class.forName` könnte dann aufgrund des Klassennamens die jeweilige Klasse ermittelt werden. Mittels `Class.newInstance` könnte dann ein Objekt der entsprechenden Klasse erzeugt werden. Für eine neue Sorte von Problemen genügen dann die Definition einer neuen Klasse und ein neuer Eintrag in der Textdatei.

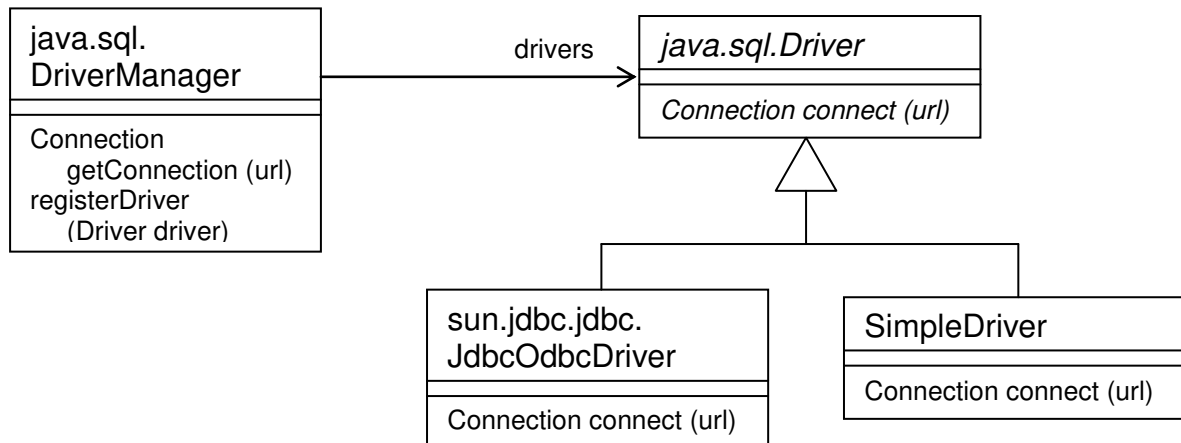
Aufgaben

Erzeugen Sie die Instanzen der `ProblemSolver` auf die oben beschriebene Weise.

4.11.3 JDBC-Driver und DriverManager

In diesem Abschnitt geht es um eine bekannte Verwendung des Patterns im Rahmen von `java.sql`. Hier sei auch auf den JDBC-Abschnitt zum Abstract-Factory-Pattern verwiesen.

Ein Klassendiagramm:



Hier das (vereinfachte!) Interface `java.sql.Driver`:

```
package java.sql;

public interface Driver {
    public abstract Connection connect (String url, Properties props)
        throws SQLException;
    // ...
}
```

Hier die (vereinfachte) Klasse `java.sql.DriverManager`:

```
package java.sql;

public class DriverManager {

    private final List<Driver>> drivers = new ArrayList<>();

    public static void registerDriver(Driver driver)
        throws SQLException {
        drivers.add (di);
    }

    public static Connection getConnection(
        String url, String user, String password)
        throws SQLException {
        Properties props = new Properties();
        props.put("user", user);
        props.put("password", password);
        for (Driver driver : this.drivers) {
            Connection con = driver.connect(url, props);
            if (con != null)
                return con;
        }
        throw new SQLException("No suitable driver", "08001");
    }
}
```

Die Klasse `DriverManager` hat eine statische Registratur, in welcher `Driver` registriert werden können (mittels der statischen Methode `registerDriver`). Die Methode `getConnection` iteriert über die installierten `Driver` und ruft jeweils die `connect`-Methode des `Drivers` auf. Dieser wird die URL der Datenbank mitgegeben. Diese Methode wird prüfen, ob der jeweilige Treiber "zuständig" ist – was sie anhand der ihr übergebenen URL entscheiden wird. Wenn ja, wird ein Objekt zurückgeliefert, dessen Klasse das Interface `Connection` implementiert – und diese `Connection` wird als Resultat von `getConnection` zurückgeliefert. Ansonsten wird `null` geliefert. Dann wird der nächste `Driver` bemüht...

Die `Driver` bilden also eine "Chain of responsibility".

Und hier eine mögliche eigene `Driver`-Klasse:

```
package simple;

import java.sql.Connection;
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class SimpleDriver implements Driver {
    static {
        try {
            DriverManager.registerDriver(new SimpleDriver());
        }
        catch (final SQLException e) {
            throw new RuntimeException(e);
        }
    }

    // ...

    @Override
    public Connection connect(String url, Properties props)
        throws SQLException {
        System.out.println("SimpleDriver.connect(" +
            url + ", " + props + ")");
        if (!url.contains("simple"))
            return null;
        return new SimpleConnection(url);
    }

    // ...
}
```

Wir die Klasse via `Class.forName` geladen, wird der statische Block der Klasse ausgeführt. Dieser erzeugt eine Instanz der `SimpleDriver`-Klasse und registriert diese in der statischen Registratur der `DriverManager`-Klasse.

Die `connect`-Methode prüft, ob die `SimpleDriver`-Klasse für die URL "zuständig" ist. Wenn ja, wird ein `SimpleConnection` erzeugt und zurückgeliefert. Ansonsten wird `null` geliefert.

Und hier eine eigene `Connection`-Klasse:

```
package simple;

import java.sql.Connection;
import java.sql.SQLException;

public class SimpleConnection implements Connection {
    private final String url;

    public SimpleConnection (String url) {
        this.url = url;
    }
    // ...
}
```

Eine Anwendung:

```
package appl;

import java.sql.Connection;
import java.sql.DriverManager;

public class Application {
    public static void main(String[] args) throws Exception {
        Class.forName("simple.SimpleDriver");
        final Connection con =
            DriverManager.getConnection("jdbc:simple//xyz");
        System.out.println(con.getClass().getName());
    }
}
```

Man beachte, dass die Anwendung die `Simple...`-Klassen nirgendwo als Bezeichner verwendet – sie verwendet nur die JDBC-Interfaces.

Die Ausgaben:

```
SimpleDriver.connect(jdbc:simple//xyz, {})
simple.SimpleConnection
```

Die `connect`-Methode des `SimpleDrivers` hat sich offenbar für die URL "zuständig" erklärt...

5

Literaturverzeichnis

5 Literaturverzeichnis

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Entwurfsmuster
Addison Wesley, 1996

John Vlissides
Entwurfsmuster anwenden
Addison Wesley, 1999

Bertrand Meyer
Object-Oriented Software Construction - Second Edition
Prentice Hall, 1997

