Design Patterns Revisited

Martin Kuhlemann
School of Computer Science, University of Magdeburg, Germany
kuhlemann@iti.cs.uni-magdeburg.de

Abstract

Design patterns are general solutions for recurring problems and used to develop flexible, reusable and modular software with Object-Oriented Programming (OOP). Prior studies have shown a lack of modularity in object-oriented design patterns. *Aspect-Oriented Programming (AOP)* aims at improving flexibility, reusability, and modularity in object-oriented designs. In a case study Hannemann and Kiczales have argued that AOP improves the implementation of GoF design patterns. *Feature-Oriented Programming (FOP)* is a new programming technique that also aims to improve the modularity in object-oriented designs. In this paper we compare OOP, AOP, and FOP in a quantiative case study of design pattern implementations. We evaluate the OOP, AOP, and FOP design pattern implementations with respect to modularity and show that FOP performs best compared to OOP and AOP.

1. Introduction

Design patterns are accepted and well known approaches to implement variable and reusable software using *Object-Oriented Programming (OOP)* [13]. Although widely accepted, design patterns lack in separating software into modules and cause *crosscutting concerns* [15].
Crosscutting concerns imply tangling, scattering and replication of source code which results in complex software [16]. Classes that include code of a crosscutting concern are closely coupled to this concern (tangling) and to the other classes that also implement this crosscutting concern. To exchange the crosscutting concern the classes that include this concern have to be replicated. Thus, software including crosscutting concerns is monolithic, hard to maintain and reuse and thus development effort increases. Crosscutting concerns are studied in ongoing research [4, 23, 27], and numerous approaches aim to tackle them on different levels of software development, e.g., during requirement engineering and others [1, 25, 8, 20]. Recently, advanced programming techniques, e.g., *Aspect-Oriented Programming (AOP)* and *Feature-Oriented Programming (FOP),* gain momentum to overcome crosscutting concerns [16, 24].
Several studies have shown the strengths of AOP and FOP [15, 14, 2]. These studies concentrated on single techniques or compared AOP and FOP qualitatively.
In this paper we compare OOP, AOP, and FOP in a quantitative case study using the Gang-of-Four (GoF) design patterns [13]. We did so to achieve a broader perspective of problems that occur frequently in software development. We show that FOP outperforms OOP and AOP with respect to modularity but also includes drawbacks.

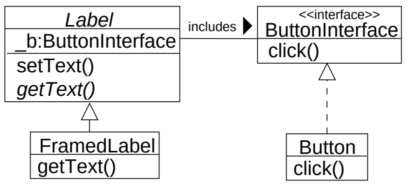

Figure 1. UML notation of OOP classes and interfaces.

2. Background

2.1 Object-Oriented Design Patterns

In OOP methods and variables are merged into *classes*. For composing classes OOP provides mechanisms of inheritance and object composition [28]. Variability of software is achieved through polymorphism of classes [7].
Object-oriented design patterns propose advantageous class arrangements for frequently recurring requirements [13]. The requirements are described by *roles* of interacting objects, e.g., if one kind of object has to observe changes of another object. If a class should play a role in one of these design patterns it is assigned to implement interfaces or to inherit classes specific to its role.

Figure 1 depicts the UML notion for OOP mechanisms [22]. The figure depicts the classes `FramedLabel` and `Button`, the abstract class `Label`, and the interface `ButtonInterface`. The class `Label` declares the method `getText`, the class `Button` defines this method. Equivalently, the method `click` is declared in the interface `ButtonInterface` and is defined in the class `Button`. Inheritance and interface implementations are denoted by arrows while inheritance implies solid arrows and interface implementations imply dashed arrows. Associations between classes are denoted by simple lines, e.g., the class `Label` includes one member of type `ButtonInterface`.

2.2 Aspect-Oriented Programming

The purpose of AOP is to modularize crosscutting concerns into *aspects* [16].
We now explain the AOP mechanisms of AspectJ[1], a popular AOP language extension for Java, that are used in our case study [15].

Pointcut and Advice. The mechanism of AOP is the extension of code implementing events that occur at runtime (so-called *join points*) [18]. The static representation of a runtime event in the source code is called *join point shadow*. Join point shadows are for example method calls, constructor calls, or member access. A *pointcut* defines a set of join points to be extended. The extension to be invoked at the join points is called *advice*.

An example for pointcut and advice (*PCA*) is given in Figure 2. The aspect `MyAspect` (Lines 12–26) extends the classes `Label` and `Button`. The pointcut `LabelChangeCall` (Line 13) refers to all

[1] http://www.eclipse.org/aspectj/

```
 1 public class Label {
 2  public void setText(){/*...*/}
 3 }
```

```
 4 public class Button {
 5  ButtonInterface _b;
 6  public void click(){
 7   /*...*/
 8   myLabel.setText("Button clicked")
 9   /*...*/
10  }
11 }
```

```
12 public aspect MyAspect {
13  protected pointcut LabelChangeCall():
      call(*  Label.*(..));
14  protected pointcut LabelChangeExec():
      execution(*  Label.*(..));
15  before():LabelChangeCall(){/*...*/}
16  before():LabelChangeExec(){/*...*/}
17
18  public String Label.Name;
19  public void Label.printName(){/*...*/}
20
21  public HashMap printer;
22  public void getPrinter(){/*...*/}
23
24  declare parents:Button implements ButtonInterface;
25  declare precedence:PriorAspect,MyAspect;
26 }
```

Figure 2. Application of call and execution advice in AOP.

statements that invoke methods of the class `Label` (call pointcut), e.g., call statements for the method `setText`. The corresponding piece of advice (Line 15) is woven into the method `click` of the class `Button` *before* (before advice) the call of the labels method `setText` is invoked (Line 8). Advice also can be applied after (after advice) or around (around advice) join points.
The advice of the pointcut `LabelChangeExec` (Line 16) refers to the body of the method `setText` (execution advice), i.e., the advice is woven into the method `setText` of class `Label` (Line 2).
While pieces of call advice, e.g., advice assigned to the pointcut `LabelChangeCall`, intercept the method caller, i.e., call advice only augments specific join points that perform the method `setText`, execution advice, e.g., advice assigned to the pointcut `LabelChangeExec`, intercepts the called object, i.e., execution advice augments all join points performing the method `setText`.

Inter Type Declaration. Inter type declarations (ITD) are methods or variables that are inserted into classes and interfaces by an aspect and thus become members of these classes and interfaces respectively. Contrary to Java conventions, AspectJ allows to introduce methods including a method body into interfaces [15].
In our example of Figure 2 the aspect `MyAspect` defines two ITD i.e., to insert the member variable `Name` (Line 18) and method `printName` (Line 19) into the class `Label`.

Aspect Fields and Methods. Aspects can contain members similar to members of an OOP class, i.e., aspects can contain methods, fields, or inner classes and interfaces. These aspect members can be invoked like methods of a class from inside the aspect, e.g., by advice, or from outside the aspect, i.e., from the classes (using the aspect method `aspectOf`). The aspect `MyAspect` includes one aspect field and one aspect method (Fig. 2, Lines 21–22).
If aspect fields and methods (AFM) are invoked through `aspectOf` and no extra pointcut mechanisms are declared (e.g., `percflow`), then every reference to the aspect members of one aspect refers to the same singleton instance of the aspect. In this case the aspect is instantiated once.

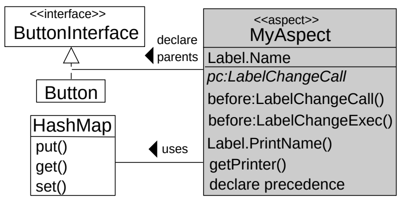

Figure 3. Graphical notation of an aspect.

Parent Declaration. Aspects can make a class to implement an interface. Furthermore, aspects can declare a class to inherit from another class.
In Figure 2 (Line 24), the aspect `MyAspect` assigns the class `Button` to implement the interface `ButtonInterface`.

Other AOP. The category *Other AOP* includes compiler warnings and errors and includes the declaration of advice precedence.
If a user defined constraint is violated by the classes, the aspect weaver can be instructed to invoke compiler warnings or compiler errors.
Precedence declarations define the ordering of advice if join point shadows are advised by more than one aspect, e.g., Fig. 2, Line 25, states that the advice of the aspect `PriorAspect` has to be applied before the advice of the aspect `MyAspect` is applied.

We depict our extended UML notion for aspects in Figure 3. The shaded element depicts the aspect `MyAspect` of Figure 2 and includes the PCA, ITD, and AFM. Pointcuts are abbreviated using `pc` while advice is abbreviated by the type of advice (e.g., before advice). We depict subclass declarations assigned by an aspect by associating the aspect to the inheritance relationship of the classes (association `declare parents`, Fig. 3).

2.3 Feature-Oriented Programming

FOP aims at feature modularity in software product lines where features are increments in program functionality, e.g., feature tracing [24, 6]. Typically, features are not implemented through one single class [26, 5] but through different *collaborating* classes and adding a feature subsequently means to introduce code, e.g., new methods, into different existing classes [24, 26]. This code of different classes associated to one feature is merged into one *feature module*. In the following selecting features of the software is equivalent to selecting feature modules. Assigning a feature to a configuration causes the new feature module to superimpose (refine) the old feature modules [6], i.e., methods and classes are added or get refined.
We systematize the mechanisms of the AHEAD Tool Suite[2], a popular FOP language extension for Java, into the categories of *Mixins*, *Method Extensions*, and *Other FOP*. Additionally, we describe the OOP technique of *Singleton* classes as a category since we used singleton classes to transform AFM into FOP.

Method Extension. FOP allows to extend methods of classes by overriding.
An example is depicted in Figure 4. The feature module BASE (Lines 1–4) includes a class `Label` that is superimposed by the refinement EXTENSION (Lines 5–12), i.e., the refinement EXTENSION superimposes the method `setText` of the class `Label`. The method `setText` of the feature module EXTENSION extends the

[2] http://www.cs.utexas.edu/users/schwartz/ATS.html

```
 1 // feature module BASE
 2 public class Label {
 3  public void setText(){/* ... */}
 4 }
```

```
 5 // feature module EXTENSION
 6 refines class Label {
 7  public void setText(){
 8   Super().setText();
 9  }
10  public String Name;
11  public void printName(){/* ... */}
12 }
```

Figure 4. Refinement of a method by a FOP refinement.

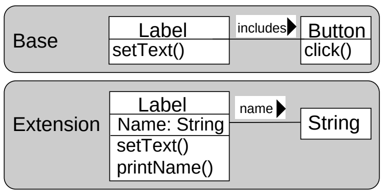

Figure 5. Refinements and classes in FOP.

setText method of the feature module BASE by invoking this superimposed method using Super (Line 8) and defining additional statements.

Mixins. Feature modules include mixin classes, that superimpose and refine other classes. Mixins are members of mixin classes, e.g., methods and member variables, that are introduced into an existing class and extend the set of members of this refined class.
In Figure 4 the new method printName and the new member Name is introduced by the feature module EXTENSION, i.e., by the mixin class Label (Lines 10–11).
We define that a subtype declaration of a mixin class is a mixin too since this property is added to a class. Thus, mixin classes can introduce additional base classes for a refined class.

Singleton. A singleton class is an idiom to limit the number of instances of a class. The singleton class is usually instantiated once and all subsequent requests to this class are forwarded to this unique instance [13].

Other FOP. This category include idioms of the arrangement of classes, the ordering of feature modules and the qualification of member variables.
All classes, that are not nested in other classes are encapsulated in feature modules. The ordering of feature modules defines the ordering of extensions for one single method. Class members qualified as limited visible, e.g., qualified as protected, cannot be accessed from classes others than the class itself and its subclasses.

Figure 5 depicts our graphical notion of FOP mechanisms that are used in Figure 4. The feature modules BASE and EXTENSION are shaded and encapsulate the classes Label and Button and a mixin class refining the class Label. (The String class of the Java-API is not implemented inside the layer but is depicted to depict special properties of the Label class.)

We refer to classes and mixin classes inside feature modules by *([X.]*)Y*, where Y is a (nested) feature module and Y is the single mixin class.

We used *mixin layers* to implement the GoF design patterns in FOP [26]. Mixin layers is one implementation technique for FOP where each refinement is figured by one class of an OOP inheritance hierarchy.

3. Goal Statement

3.1 What Do We Adress?

AOP and FOP provide benefits compared to OOP but have difficulties and strengths [3]. In this paper we aim to compare OOP, AOP, and FOP implementations with respect to modularity.

3.2 Experimentation Methodology

3.2.1 Criteria

We compare the design pattern implementations with respect to the properties of modularity, i.e., *cohesion* and *variability*, and thus we follow existing studies of OOP and AOP [15, 14]. We define these properties as follows:

Cohesion. An aggregate definition, e.g., a package, of different program changes, e.g., the introduction of different classes or methods into the software, can be referred to by a name and is therefore cohesive [17]. The named module, e.g., a package, can be exchanged and reused and thus development effort decreases.
Cohesive modules rarely reference to other modules and are thus loosely coupled to other modules.
Our definition of cohesion is similar to the definition of *Locality* of Hannemann et al. and *Cohesion* of Garcia et al. [15, 14].

Variability. If features of a modularized software shall be able to change flexibly, the modules of the software have to be composed in many different ways. Modules that are loosely coupled are prerequired [17]. Consequently, modules can be exchanged easily.
Tangling of code of different concerns of the software causes *close* coupling of modules resulting in invariant, complex and monolithic software [16].

Our definition of variability corresponds to the criteria *Composition Transperency* of Hannemann et al. [15]

In addition, Hannemann et al. used the criteria *Reusability* and *(Un)pluggability* to evaluate the aspect-oriented design pattern implementations [15]. We do not use these criteria because we argue them to be imprecise and not significant.

3.2.2 Schedule of Comparison

We adopt the methodology of Hannemann et al. and Garcia et al. to evaluate programming paradigms [15, 14]. Both studies compared OOP and AOP on the base of a case study of design pattern implementations. To analyze many diverse applications we reimplemented the 23 GoF design patterns in FOP and adopt the OOP and AOP implementations. For different design patterns we implemented alternative FOP implementations (up to 7 per design pattern) resulting in 50 different FOP implementations. For comparison we choose the implementation that is close to the AOP counterpart.

We compare the different implementations of the design patterns by repeating the following schedule:

1. We review the aim of the pattern.
2. We give an explanation of the OOP, AOP, and FOP implementations each followed by a discussion of the specific pros and cons.
3. We compare the OOP, AOP, and FOP implementations based on the criteria given in Section 3.2.1.
4. We give a short summary of specific difficulties and strengths of the OOP, AOP, and FOP implementations that were captured

```
public interface ComponentFactory {
 public JLabel createLabel();
 ...
}
```

Figure 6. Type limitation through method declarations in the OOP Abstract Factory implementation.

during the evaluation but are not relevant for the analyzed criteria.

As a summary we give a table that aggregates and balace the results of the evaluation. A "+" depicts that the technique performs well with respect to the criterium while "-" depicts the lack of the technique regarding that criterium compared to the other techniques. "0" depicts the neutral evaluation regarding the criterium and compared to the other techniques.

4. Case Study

In this section we evaluate the design patterns implemented in OOP, AOP, and FOP in detail.

Hannemann et al. use Java[3] to implement design patterns in OOP and use AspectJ to implement the aspect-oriented counterparts. We use the AHEAD Tool Suite for implementing the patterns in FOP.

4.1 The Abstract Factory Design Pattern

4.1.1 Intention

Provide an interface for creating families of related or dependent objects without specifying their concrete classes [13].

4.1.2 Implementation

OOP solution. Hannemann et al. applied the pattern to create different kinds of one graphical user interface (GUI) [15].

Each factory class creates GUI elements of different kinds, e.g., buttons or labels, and of different properties for each kind of GUI element, e.g., one factory object creates framed or regular elements of different kind (e.g., `Button`). All GUI elements (e.g., of type `Button` or `Label`), that are created by one factory class (e.g., `FramedFactory` or `RegularFactory`), have compatible properties, e.g., all elements are framed or all elements are regular. The properties of GUI elements created by different factories differ and may be incompatible. Each GUI is created using one factory and thus all elements of one GUI have compatible properties, the elements of different GUI may have different properties because they were created by different factories. Hence, the choice of the factory class implies the properties of the graphical elements that are created. Different factory classes, i.e., `FramedFactory` and `RegularFactory`, can be exchanged with respect to the common interface `ComponentFactory`. If a referenced `FramedFactory` object is replaced by a `RegularFactory` object the GUI elements created subsequently are framed instead of regular and vice versa without affecting a client that creates GUI.

Advantages Exchanging factory objects of different type (e.g., `RegularFactory` and `FramedFactory`) does not affect the client that refers to the interface `ComponentFactory`. A compatible configuration of properties for different GUI elements to be built, e.g., whether all should be framed, is encapsulated inside one graphical factory (e.g., `FramedFactory`).

Disadvantages The factory classes determine the type for each kind of GUI element, e.g., a button, that can be created by the factory. Consequently, all GUI elements that should be created by the factory have to be of the type that is referred to by the factory class for the according kind of GUI elements, e.g., the method `createLabel` of the factory `ComponentFactory` limits the type of possibly created label objects to be subtype of the class `JLabel` (Fig. 6, Line 2).
Different factories may create the same GUI element classes and thus introducing code replication.
If the implementation of the methods `createLabel` or `createButton` should vary for all factory classes in the same way either all classes or the common superclass has to change. This introduces code replication if both variants of the implementation should be available.

[3] http://java.sun.com/

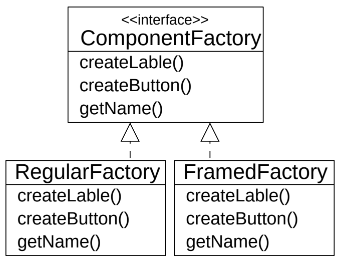

Figure 7. Abstract Factory through OOP.

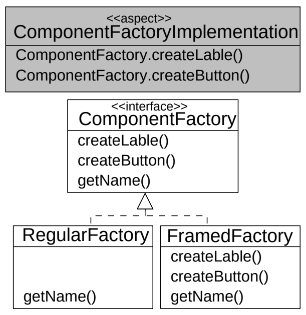

Figure 8. Abstract Factory through AOP.

AOP solution. In the AOP implementation the factory class can create GUI elements, e.g., of type `Button`, differently without conditional statements or subclasses of the factory class. The methods that create the GUI elements are detached into the aspect `ComponentFactoryImplementation` (Fig. 8) and are introduced on demand.

Advantages The advantages of the OOP implementation hold for the AOP implementation. The interface `ComponentFactory` can be extended by methods without code replication of the class or its subclasses. Thus, variant implementations of the methods `createLabel` or `createButton` can be varied for all classes, e.g., `FramedFactory`, homogenously without introducing code replication.

Disadvantages The AOP implementation does not work without the aspect `ComponentFactoryImpl` because the methods `createLabel` and `createButton` are declared in the interface `ComponentFactory` but never implemented by its subclasses. This arises the cognitive distance.
The variable composition of the factory class may hamper compatibility of GUI elements applied to that factory class.

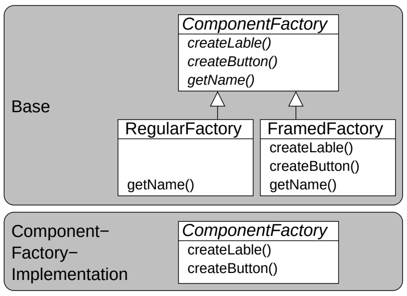

Figure 9. Abstract Factory through FOP.

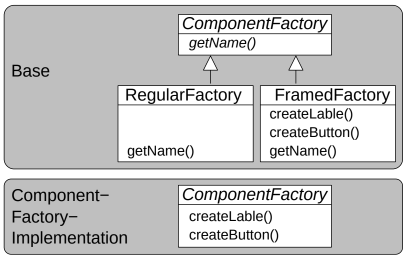

Figure 10. Abstract Factory through FOP reduced virtual methods.

FOP solution. We present 2 solutions for the pattern Abstract Factory: solution A is close to the AOP implementation (Fig. 9). Variable methods of the factory class, e.g., `createLabel`, are detached into the feature module COMPONENTFACTORYIMPLEMENTATION. Since we extend the interface `ComponentFactory` subsequently using method definitions we transformed it into the *abstract class* `ComponentFactory`.
Solution B is depicted in Figure 10. The different implementations of the methods `createLabel` and `createButton` are transferred completely to the feature module COMPONENTFACTORYIMPLEMENTATION, i.e., no declarations are left in the feature module BASE.

Advantages The advantages of the OOP and AOP implementations hold for the FOP solution A and B. Solution B works without the `componentFactoryImpl` refinement because no method is declared without being implemented in the feature module BASE. This decreases the cognitive distance.

Disadvantages The variable composition of the factory class may hamper compatibility of GUI elements provided by the configured class.

4.1.3 Discussion

Cohesion. Two issues have to be analyzed with respect to cohesion:

- In the OOP implementation the code associated to variable method implementations (e.g., to the method `CreateLabel`) is closely coupled to the code of the factory classes that is not variant, e.g., the method `getName`. In the AOP implementation the variant methods code of the factory classes (method `createLabel`) is detached into the aspect `ComponentFactory-Implementation` but modules of the base program and the module of the variant aspect are not separated. In the FOP implementation the the different method implementations are separated, i.e., into the class `ComponentFactory` and the according mixin class `ComponentFactoryImplementation.ComponentFactory`. Furthermore, the FOP implementation separated the modules that are non-variant, e.g., `TextCreator`, from the modules that are variant, e.g., the mixin class `Creator-Implementation.Creator`.
- In the OOP implementation compatible objects to be created by a factory object are defined inside one class. In the AOP and FOP implementations compatible compositions of methods that create GUI elements are scattered across the factory class and the aspect and mixin class `ComponentFactory` respectively.

Variability. The OOP implementation does not allow to exchange the definitions of the methods `createLabel` and `createButton`. The AOP implementation allows to exchange the implementation of the factory methods (with respect to the interface `Component-Factory`) without causing code replication. The FOP implementation equivalently does not imply code replication if the factory method implementations should be exchanged for all subclasses.

Criteria	OOP	AOP	FOP
Cohesion	0	0	+
Variability	0	+	+

Table 1. Evaluation of the pattern Abstract Factory

4.1.4 Summary

A summary of the evaluation of the Abstract Factory design pattern is given in the Table 1.

4.2 The Adapter Design Pattern

4.2.1 Intention

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of the incompatible interfaces [13].

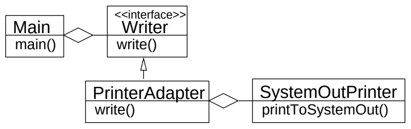

Figure 11. OOP implementation of the Adapter pattern.

4.2.2 Implementation

OOP solution. Hannemann et al. applied the Adapter pattern to invoke incompatible printer objects, e.g., of type `SystemOutPrinter` (Fig. 11), from the main method. The print request is intercepted and forwarded to the according printer class by an adapter object (e.g., of type `PrinterAdapter`). This adapter object is invoked instead of the incompatible printer object of type `SystemOutPrinter`. The adapter object adapts the request to fit the incompatible printer object interface (`SystemOutPrinter`). Different adapter classes can be exchanged with respect to the `Writer` interface and thus different incompatible printer objects can be used.

Advantages The usage of incompatible printer objects, e.g., of type `SystemOutPrinter`, does not affect the calling main method. They can be used polymorphically. The adaption of the print request to fit the interface of the incompatible printer class `SystemOutPrinter` is modularized into the class `PrinterAdapter`. The printer implementations classes, e.g., `SystemOutPrinter`, do not have to implement a specific interface to be used polymorphic by the main method.

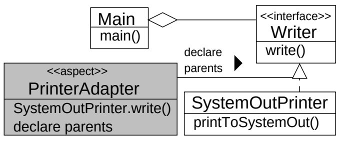

Figure 12. Adapter pattern in AOP.

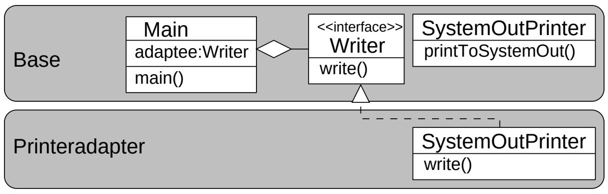

Figure 13. Adapter pattern in FOP.

Disadvantages The adapter class `PrinterAdapter` only can invoke *public* members of the `SystemOutPrinter` class to perform the print request of the main method.
The adapting method `write` of the `PrinterAdapter` class has to be bound dynamically to enable the polymorphic exchange of different adapter objects – this corrupts performance and resource consumption [10].

AOP solution. In the AOP implementation the adapting method `write` is introduced into the incompatible printer class (`SystemOutPrinter`) by the aspect `PrinterAdapter` (Fig. 12). The incompatible printer class `SystemOutPrinter` is adapted to be usable by the method `main`. The introduction of different adapting methods, e.g., `write`, into the incompatible printer class `SystemOutPrinter` prevents changes of the class `SystemOutPrinter` to implement a specific interface, e.g., `Writer`.

Advantages The advantages of the OOP implementation also hold for the AOP implementation. The adapting function introduced into the class `SystemOutPrinter` via ITD has full access to members of that class (based on the open class mechanism of AOP [21]). Therefore, the adapting method `write` can invoke protected and private methods of the incompatible `SystemOutPrinter` class.

Disadvantages The essential code of OOP classes does not work without the aspect `PrinterAdapter`.
The method `write` has to be bound dynamically although only one variant may be present at runtime, thus corrupting performance and resource consumption [10].
If the class `SystemOutPrinter` should be adapted to more than one incompatible caller (e.g., the method `main`) another aspect may introduce another `write` method thus causing compiler errors.

FOP solution. We present 2 solutions for the adapter pattern: solution A is close to the AOP implementation (see Figure 13). We introduce the translating method `write` into the `SystemOutPrinter` class and thus the method `main` uses that introduced method of the `SystemOutPrinter` class.
Solution B is close to the OOP implementation (see Fig. 14), i.e., we introduce the translating method into the adapter class `Writer`. Solution B is applicable if the concrete writer does not have to vary at runtime.

Advantages Solution A keeps the advantages that hold for the OOP implementation.

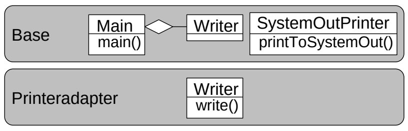

Figure 14. Alternative implementation of the Adapter pattern in FOP.

Criteria	OOP	AOP	FOP
Cohesion	0	0	0
Variability	0	+	+

Table 2. Evaluation of the pattern Adapter

The translating method `write` has advanced access to the members of the `SystemOutPrinter` class. If mixin layers are used to implement FOP each refinement is implemented as subclass and thus access is prevented to private members of the superclass, i.e., previous refinements. If FOP is implemented using Jampacks the `write` method can access *all* members of the `SystemOutPrinter` class that are introduced by prior refinements.

Solution B decreases the number of virtual methods because the adapting method `write` is inserted into the writer class statically bound which improves performance and resource consumption.

Disadvantages The basic code of OOP classes does not work without the refinement `PrinterAdapter`.
If the `SystemOutPrinter` class has to be adapted in solution A to be applicable for different clients different feature modules, e.g., PRINTERADAPTER, may introduce different versions of the adapting `write` method that override each other.
Solution A demand for the `write` method to be bound dynamically thus corrupting performance and resource consumption.

In solution B the class `Writer` only can use public members of the `SystemOutPrinter` class.

4.2.3 Discussion

Cohesion. The techniques OOP, AOP, and FOP are equivalent regarding the cohesion in the analyzed implementation of the adapter pattern. That is because the *complete* class `PrinterAdapter` was transformed into an aspect in the AOP implementation and into a refinement in the FOP implementation respectively.

Variability. In OOP exchanging the adapter, e.g., `PrinterAdapter`, does affect the `main` method since it has to instantiate a different class. In the AOP and FOP implementations the adapting method `write` can be exchanged flexibly without changes to the `main` method, because this method does not refer to the adapter class but directly to the `SystemOutPrinter` class.

4.2.4 Summary

A summary is given in Table 2.

4.3 The Bridge Design Pattern

4.3.1 Intention

Decouple an abstraction from its implementation so that the two can vary independently [13].

4.3.2 Implementation

OOP solution. Hannemann et al. applied the pattern to define composes and complex operations, e.g., `drawGreeting` or

`drawText` of a Screen class (Fig. 15), based on different versions of primitive operations, like `printLine` and `printDecor`. These primitive operations are detached to the classes `CrossCapitalImplementation` and `StarImplementation`. The different implementations of the primitive operations including classes (`CrossCapitalImplementation` and `StarImplementation`) can be exchanged with respect to the interface `ScreenImplementation` and thus the definition of the composed operations can be reused. The composed operations, e.g., `drawGreeting`, are defined in the classes `InformationScreen` and `GreetingScreen` thus extending the interface of `Sceen` objects.[4]
The interface of a complex object is implemented using an primitive interface that forwards calls to different implementing classes.

Advantages The interface of `Screen` objects and its implementation can be extended independently. Changing the implementation of a `Screen` object at runtime does not affect clients, like the `main` method.

Disadvantages If the class hierarchy implementing the primitive operations, e.g., `ScreenImplementation`, should be exchanged, i.e., the static type of the classes implementing the primitive operations should be exchanged, the class `Screen` has to be changed or extended – this either demands for code replication or invasive changes. Consequently, classes that implement the primitive operations, e.g., `printLine`, have to implement the interface `ScreenImplementation`.

The primitive operations, e.g., `printDecor`, inside the `Screen` class forward requests to the classes implementing the primitive operations, e.g., to the class `StarImplementation`, thus decreasing performance. The methods implementing the primitive operations inside the `StarImplementation` class have to be bound dynamically to be polymorph regarding the interface `ScreenImplementation` thus corrupting performance and resource consumption.

AOP solution. The AOP implementation extracts the primitive operations, e.g., `printDecor`, (the forwarding methods) into the aspect `AbstractionImplementation` (Fig. 16). Additionally the class to implement the primitive operations, e.g., `ScreenImplementation`, is defined by the aspect.

Advantages The advances of the OOP implementation also hold for the AOP implementation. Different classes, e.g., `CrossCapitalImplementation`, that implement the primitive operations, like `printDecor`, do not have to be subtype of a common interface.

Disadvantages The program does not work without the aspect.

FOP solution. We present two solutions for the Bridge design pattern: solution A (Fig. 17) is close to the AOP implementation, that is, the methods that forward the primitive operations (i.e., `Screen.drawText`) are detached into the feature module ABSTRACTIONIMPLEMENTATION. Additionally, the mixin class of this feature module defines the class that implements the primitive operations, e.g., `ScreenImplementation`.

Solution B is depicted in Figure 18. Since no method implementation and member reference is left inside the `Screen` class, we omit this class but transformed the `Screen` mixin into the new super-class of the composed operation classes (e.g., `InformationScreen`)

Advantages The advantages of the OOP *and* AOP implementations hold for the FOP solution A.

[4] To invoke methods of the $GreetingScreen$ class that are not declared in the $Screen$ class casts have to be performed.

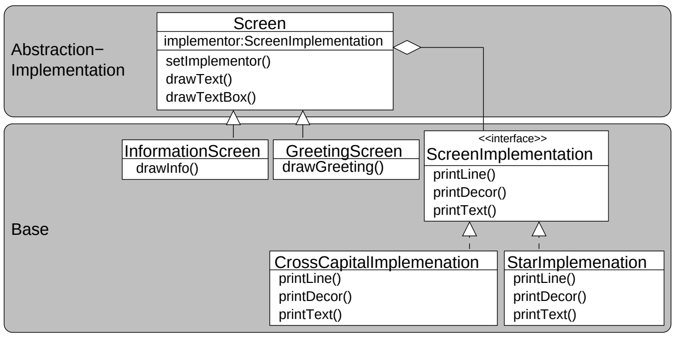

Figure 18. Complexity reduced implementation of the Bridge pattern in FOP.

Criteria	OOP	AOP	FOP
Cohesion	-	0	+
Variability	0	+	+

Table 3. Evaluation of the pattern Bridge

Solution B reduces the number of classes by omitting the abstract class `Base.Screen` and the number of methods that are bound dynamically, e.g., `drawText`, thus improving performance and resource consumption.

Disadvantages The basic implementations (inside the feature module BASE) do not work without applying the refinement ABSTRACTIONIMPLEMENTATION.

4.3.3 Discussion

Cohesion. In the OOP implementation the code regarding the definition of complex operations, e.g., `drawText`, is coupled with code regarding variant implementations of the primitive operations. The AOP implementation separates the variant class members from the essential but lacks in separating the modules that implement variant and invariant behavior respectively. In the FOP implementation the variant primitive operations are decoupled from the essential methods by detaching and the modules implementing variant and invariant operations are separated.

Variability. In the OOP implementation only classes can be used to perform the primitive operations that implement the needed methods and that are subtype of the interface `ScreenImplementation`.
In the AOP and FOP implementation every class that provides according method definitions can be assigned to perform the primitive operations – they do not have to implement a common interface.

4.3.4 Summary

Since the system does not work without an aspect, the aspect must have been planned and prepared from the beginning. Therefore it is questionable whether it is worse to transform the inheritance tree in OOP design instead of using the aspect that is hard to trace.
An overview over the evaluation of the Bridge pattern is given in Table 3.

4.4 The Builder Design Pattern

The Builder pattern is applied to create different complex documents.

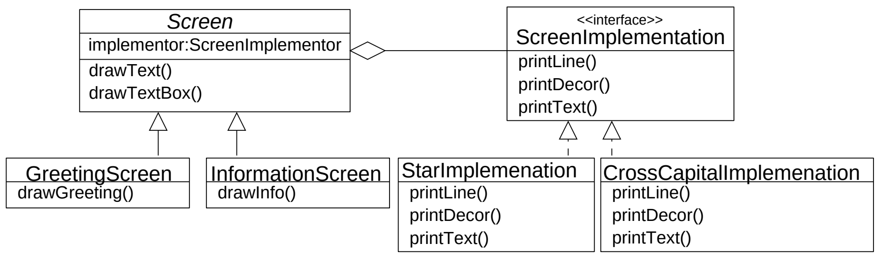

Figure 15. OOP implementation of the Bridge pattern.

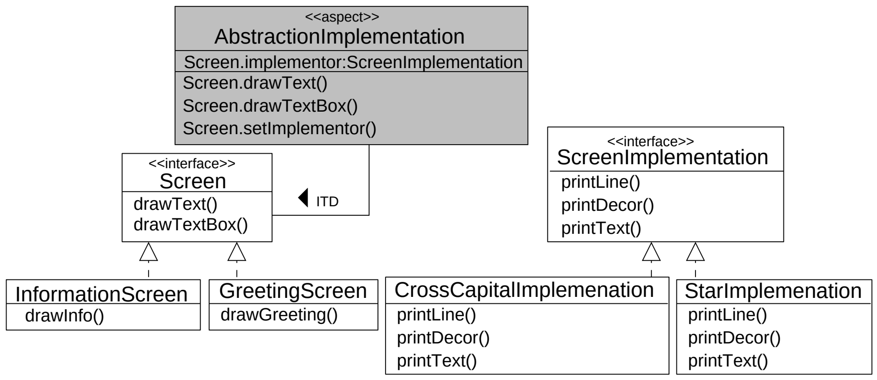

Figure 16. AOP implementation of the Bridge pattern.

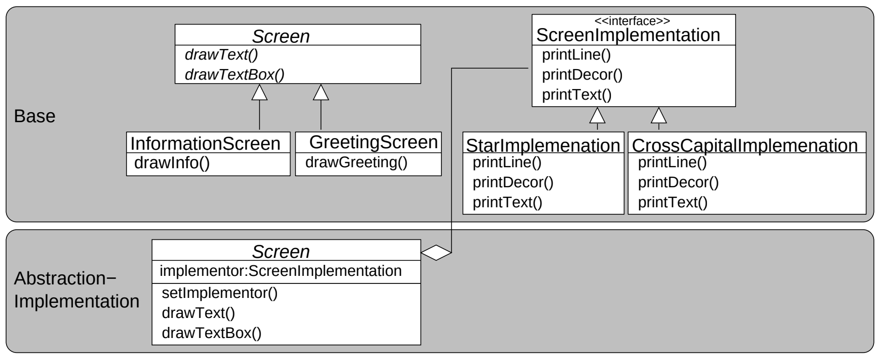

Figure 17. Direct implementation of the Bridge pattern in FOP.

4.4.1 Intention

Separate the construction of a complex object from its representation so that the same construction process can create different representations [13].

4.4.2 Implementation

OOP solution. Different complex objects (text and XML documents) are created/initialized by concrete Builders that compose the documents out of parts, e.g., attribute names and values. The concrete Builders implement the common abstract class `Creator` which is referenced by a client that uses the Builder classes. The complex document to build is hold in a document representation member of type `String`.

Figure 19 depicts the concrete Builder classes `TextCreator` and `XMLCreator`. These classes can vary with respect to the common abstract class `Creator` without a client, that only references the `Creator` class, has to change. Hence, the client is unaware of the concrete format of the complex document it is building. A member variable of type `String` (Fig. 20, Line 2) holds the document

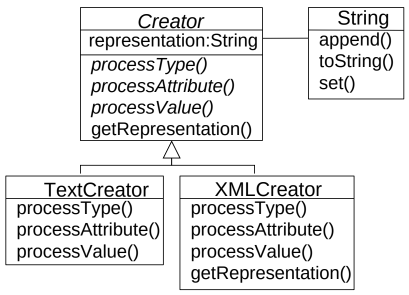

Figure 19. OOP design of the Builder pattern.

```
 1 public abstract class Creator {
 2  protected String representation;
 3  public abstract void processAttribute
      (String  newAttribute);
 4 }
```

```
 6 public class TextCreator extends Creator {
 7  public void processAttribute(String  newAttribute){
 8   representation.append("Its "+newAttribute+" is ");
 9  }
10 }
```

Figure 20. Abstract and concrete Builder classes in OOP.

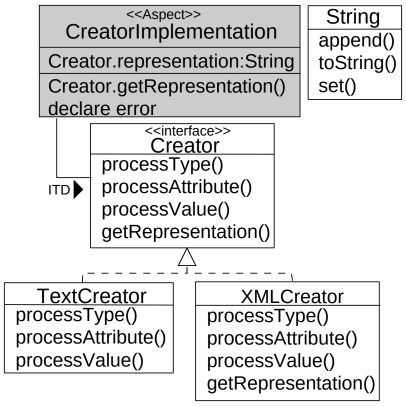

Figure 21. AOP design of the Builder pattern.

representation in XML or text format and is manipulated directly by the concrete Builders (e.g., `TextCreator`, Line 8).[5]

Advantages The concrete Builder classes, e.g., `XMLCreator`, can be exchanged at runtime due to the uniform superclass `Creator`. Thus, different complex documents (e.g., XML or text documents) can be built by a client that only is aware of the uniform abstract class `Creator`.

Disadvantages The document representation member – here of type `String` (cf. Fig. 19) – is fixed. If the type of the document representation member should vary, e.g., the member should become of type `File`, and both variants should be available, the abstract class `Creator` has to be extended or manipulated and thus code replication of the `Creator` class and subclasses, e.g., `XMLCreator`, is introduced.

AOP solution. Hannemann et al. introduce additional flexibility to the OOP solution of the Builder pattern by detaching the document representation member from the `Creator` class and encapsulating it into an aspect (Fig. 21). Figure 22 depicts an excerpt of the aspect `CreatorImplementation` that introduces the member of the document representation into the interface `Creator` via IDT (Line 2). Furthermore, an error declaration shields the new document representation member from access by classes others than the `Creator` class and subclasses (Lines 3–7).

Advantages The advantages of the OOP implementation hold for this AOP implementation. The static type of the document representation member can vary by applying different aspects each

```
1 public aspect CreatorImplementation {
2  public String Creator.representation;
3  declare error:
    (set(public  String  Creator+.representation)
4       || get(public String Creator+.representation))
5    && ! (within(Creator+)
6       || within(CreatorImplementation)):
7   "variable result is aspect protected. Use
     getResult() to access it";
8 }
```

Figure 22. Access limitation to class members in AOP.

```
1 refines class Creator{
2   protected String representation;
3 }
```

Figure 23. Member shielding in the FOP implementation.

introducing this member using a different type for it.
The document representation member is hidden by the interface `Creator`. Hence, exchanging the document representation member does not affect the clients, that use Builders to create documents. (The Builders public interface `Creator` does not change.)

Disadvantages Classes that are used for the document representation member, e.g., `String` or `File`, have to fulfill different properties. The builder classes `TextCreator` and `XMLCreator` call methods of the document representation member directly, e.g., the method `append` (the called methods are depicted in the `String` class of Figure 21)[6]. The Builder classes require these methods to be implemented in every class that is used for the document representation member but they do not reference an explicit interface class. If other classes, like `File`, should be used to hold the document representation, then they have to provide this implicitely required interface.

FOP solution. We present two FOP solutions: solution A is close to the AOP implementation (Fig. 24). The document representation member is introduced via a mixin. To shield the document representation member from external access, the according member variable is qualified as inaccessable for classes others than the `Creator` classes and subclasses, i.e., it is qualified as `protected` (Fig. 23, Line 2).
Figure 25 depicts solution B that allows to exchange the concrete format of the document, i.e., text or XML, only at compile time but not at runtime. We introduce the functions to build text or XML documents directly into the `Creator` class using mixins. Hence, the interface of the class `Creator` is not declared explicitly by an interface class but is assembled at compile time by superimposing the mixin classes of the feature modules.

Advantages Solution A allows to replace the concrete `Creator` classes (e.g., `XMLCreator`) at runtime because they implement a common interface, i.e., solution A allows to replace the complex document, that is build, at runtime.
The type of the document representation member, e.g., `String` or `File`, can vary without replicating the `Creator` class or its subclasses because the member is detached into a mixin class.
Solution B decreases the overhead that is needed for runtime configuration and that is caused by dynamic binding, e.g., for C++ implementations. Java implementations are not improved

[5] We applied minor renamings to the implementation to improve comprehensibility. [15] implied that the document representation member fulfills different $String$ operations.

[6] To improve the comprehensibility we renamed some operations.

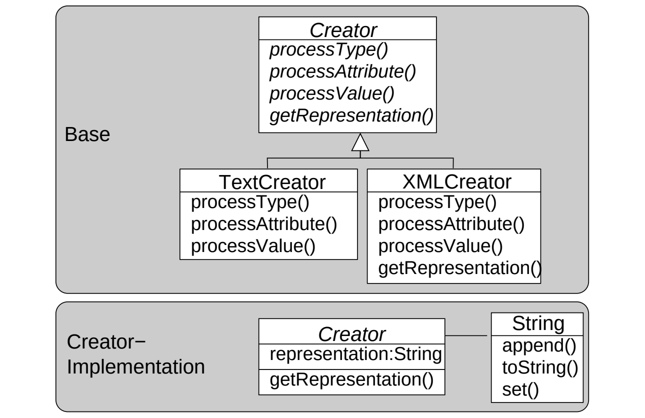

Figure 24. Straight forward transformed FOP design of the Builder pattern.

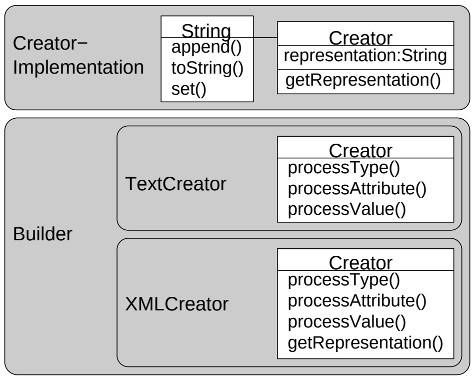

Figure 25. Static composition of the Builder class.

with respect to performance since in Java all methods are bound dynamically [9].

Disadvantages In solution A the required properties of the classes, that are used for the document representation member, are given implicitly (similar to the AOP implementation), e.g., the method `append` is required by the Builder classes.
Solution B does not allow to replace the concrete Builder implementation, i.e., the format of the complex documents that are build (text or XML), at runtime.

4.4.3 Discussion

Cohesion. To evaluate the different implementations we have to consider different aspects:

- In OOP the initialization of the complex document is coherently implemented in the class `Creator`. In the AOP implementation the document representation member is detached and the code regarding the Builder definition (class `Creator`) is scattered across the class `Creator` and the aspect `CreatorImplementation`. In the FOP implementation the document representation member of the `Creator` class is detached into the refinement CREATORIMPLEMENTATION, i.e., the code associated to the `Creator` class is scattered across the feature modules BASE and CREATORIMPLEMENTATION.
- The OOP implementation couples code of different concerns, i.e., the variant types of the document representation member (e.g., `String` or `File`) and the Builder methods, that use this member but are invariant, into one class which decreases cohesion. In the AOP implementation the code associated to variant types of the document representation member is detached from the class `Creator` into the aspect `CreatorImplementation`, i.e., but the variant extending aspects are not separated from the code that is not variant, e.g., `TextCreator`. In the FOP implementation code regarding variant types of the document representation is not coupled but cohesively separated in feature modules.

Criteria	OOP	AOP	FOP
Cohesion	0	0	+
Variability	0	+	+

Table 4. Summarized evaluation of the Builder design pattern.

Variability. In the OOP implementation the type of the document representation member in the class `Creator` is fixed. Exchanging the member type leads either to code replication or invasive changes and thus worsen reusability and complexity [12].
AOP and FOP improve variability through decoupling the Builder classes (`TextCreator`, `XMLCreator`) from their document representation members. In the following document representation members of different types, e.g., `String`, can be used by the XML and text Builder classes

4.4.4 Summary

In the code documentation Hannemann et al. admit that no advantage associated to the modularity is provided by the AOP implementation. A summary of our evaluation of the Builder implementations is given in Table 4.

4.5 The Chain of Responsibility Design Pattern

4.5.1 Intention

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it [13].

4.5.2 Implementation

OOP solution. Hannemann et al. used the pattern to allow different graphical objects (handlers) to perform actions when a button is clicked. We present two solutions. We refer to the approach of Hannemann et al. as solution A, see Figure 26. A click on a button is coded into an object of the type `Click` that is forwarded to different handler objects, e.g., of type `Button`, `Panel`, or `Frame`. After analyzing the overgiven `Click` object, e.g., using the `Click.hasCtrl` method, each invoked handler object decides either to perform actions on its own or to forward the `Click` object to the next possible handler. Hence, each handler, e.g., of type `Frame`, refers to a

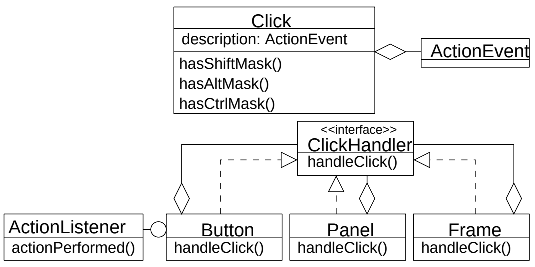

Figure 26. OOP implementation of the Chain of Responsibility pattern.

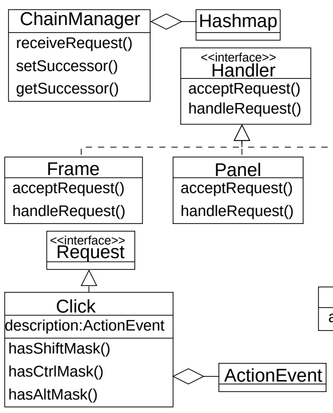

Figure 27. Alternative OOP implementation of the Chain of Responsibility pattern.

succeeding handler object potentially performing actions based on this click – consequently a recursive chain of handler objects results. The succeeding handler object, e.g., of type `Panel`, of each preceeding handler object, e.g., of type `Frame`, can be exchanged with respect to the interface `ClickHandler`, i.e., the chain can be adapted.[7]

We implemented an alternative solution B that uses a hashmap managed by a singleton class to define the succeeding handler object of a given handler object (Fig. 27), i.e., to determine the `Panel` object as succeeding handler of the `Frame` object.

The graphical elements that potentially handle a click also can be referred to by a iterative list inside the `Button` class (not depicted).

Advantages Objects, that can perform actions in response of a click on a button, e.g., `Frame`, can be assigned flexibly to the clicked object of type `Button`. Hence, the class of the objects triggering the chain (`Button`) is not affected and does not have to change when the chain differs.

Disadvantages There is no guaranty, that any handler object assigned to the chain of the button perform actions after a click on the button at all.
The classes of the objects assigned to the chain have to provide the chain-specific methods, i.e., `handleRequest` and `acceptRequest`, and have to be subtype of the interface `ClickHandler`.
In solution A the handler classes, e.g., `Frame`, can only be used as chain elements since the succeeding handler reacting on clicks has to be defined at instantiation time. This drawback is not present in solution B since the successor is stored coherently in the `ChainManager` class.

AOP solution. In the AOP implementation the members of the handler classes performing the evaluation (`acceptRequest`) of clicks and the actions to perform at clicks (`handleRequest`) are detached from the handler classes and are merged into the aspect `ClickChain` (Fig. 28). That is, the aspect `ClickChain` introduces the methods associated to the chain, e.g., `acceptRequest` and `handleRequest`, into the handler classes, e.g., `Frame`. The order in which the handler objects are called to perform actions based on a click is defined in a hashmap that is a field of the aspect `ChainOfResponsibilityProtocol` that is filled with handler objects, e.g., of type `Frame`, and their succeeding handler

[7] Furthermore, it is possible to declare the member associating the succeeding handler inside an *abstract class* `ClickHandler` – that improves cohesion (not depicted).

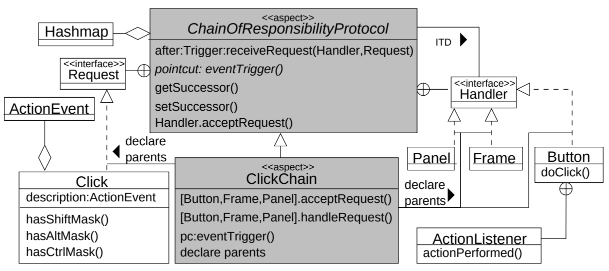

Figure 28. AOP implementation of the Chain of Responsibility pattern.

objects. The hashmap is manipulated using aspect methods, e.g., `setSuccessor`, i.e., the order of chain handlers is determined by manipulating the hashmap.

The invokation of the chain is applied using PCA (`eventTrigger`) that provides the clicked button object and the `Click` object.[8]

Advantages The advantages regarding the OOP implementation are kept. Every class, e.g., `Panel`, can be assigned to be handler of a click – the required methods and inheritance declarations are inserted by the `ClickChain` aspect. The management of the handler objects in the chain is merged into the `ChainOfResponsibilityProtocol` aspect, that is, the declaration of the respective successor of a handler is not scattered across the handler classes, e.g., `Frame`.

Disadvantages There is no guaranty that actions are performed at all in response of a click by any assigned handler object.

FOP solution. We present three different FOP implementations for that pattern. Solution A is close to the AOP implementation, see Fig. 30. We extend the class of the clicked objects (`Button`) to invoke the chain, i.e., the action listener defined inside the class `Button`. To extend the anonymous class `Actionlistener` (Fig. 29) we have to extract the class into the class `MyActionListener`, we refined this class to invoke the reacting handler objects.

The order of handler objects to be invoked after a click is defined by manipulating the hashmap that stores the succeeding handler objects for each invoked handler.

In solution B we extend the method that is called after the button is clicked (method `Button.doClick`), i.e., the `call` pointcut turns into an execution pointcut of that method effectively (Fig. 31).

Solution C is an extension of solution A. For solution C we detach the method evaluating `Click` objects (`acceptRequest`) and the definition of actions to be performed subsequently (method `handleRequest`) of every class into feature modules, e.g., `chain.clickChain.chainElements.Frame`, see Figure 32. Additionally, the invocation of the chain of handlers is detached into the feature module EVENTTRIGGER and can be exchanged.

Advantages The advantages of the OOP implementation also hold for the FOP implementation. Additionally, every class can be assigned to invoke actions when a button is clicked – the required methods, like `acceptRequest` and `handleRequest`, are introduced subsequently by the refinement CLICKCHAIN. The chain is created by ordering the handlers using a hashmap. In solution C the chain specific methods, e.g., `handleRequest`, for all classes, e.g., `Panel`, can vary independently.

[8] The button object clicked and the $Click$ object are received by evaluating the call of the $ActionListener$ of the button to a hook method $doClick$.

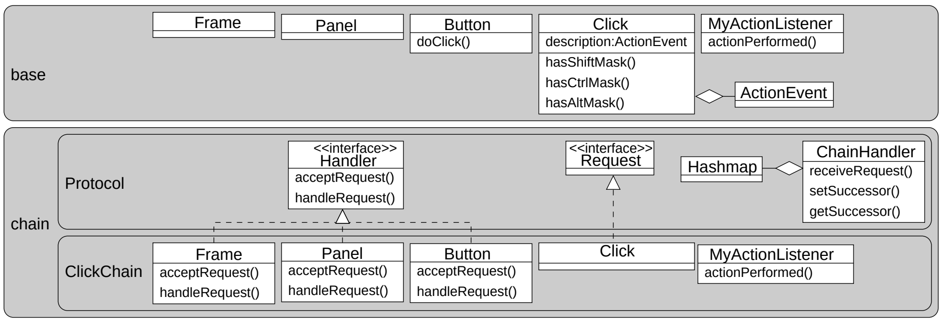

Figure 30. FOP implementation of the Chain of Responsibility pattern.

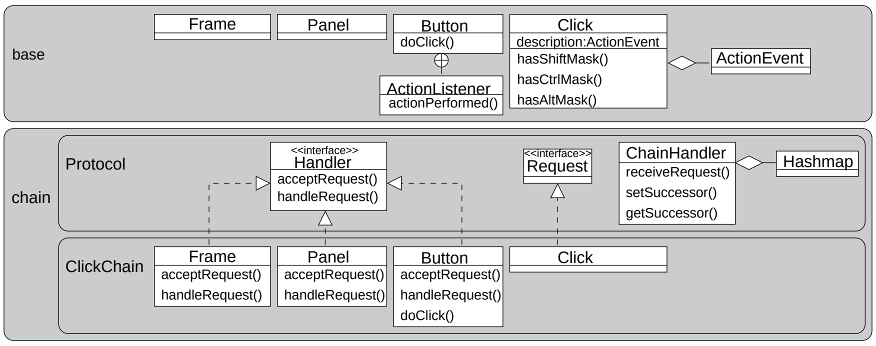

Figure 31. FOP implementation of the Chain of Responsibility pattern applying execution advice.

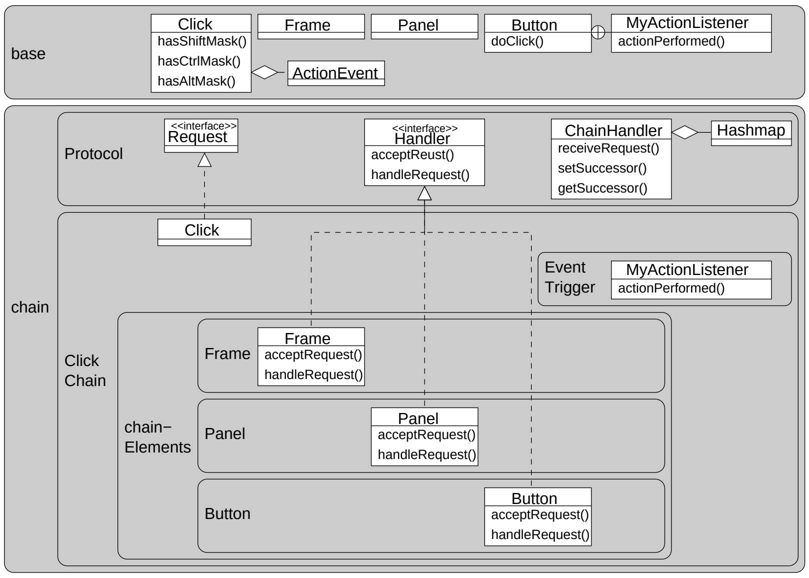

Figure 32. Finegrained FOP implementation of the Chain of Responsibility pattern.

```
public class Button extends JButton implements
  ClickHandler {
 public Button(String label, ClickHandler successor)
   {
  super(label);
  this.successor = successor;
  this.addActionListener( new ActionListener() {
   public void actionPerformed(ActionEvent ae) {
    handleClick(new Click(ae));
   }
  });
 }
}
```

Figure 29. Anonymous visitor class in the AOP implementation that triggers the chain.

Disadvantages As in OOP and AOP, there is no guaranty that any object, that is assigned as handler to the chain, ever perform actions when a button is clicked.

4.5.3 Discussion

Cohesion. To evaluate the implementations with respect to cohesion, we have to consider two issues:

Cohesion of the chain: The OOP approach of the pattern scatters the code associated to the chain, e.g., method `handleClick`, across all handlers, e.g., of type `Frame`, that potentially invoke actions after a button is clicked. The AOP implementation merges the pattern into the `ChainOfResponsibilityProtocol` and `ClickChain` aspects, hence, it stays scattered. In the FOP implementation the pattern specific code is merged into the feature module CHAIN.

Cohesion of the graphical element implementations: The OOP implementation merges the code associated to one class. AOP scatters the code regarding one graphical element, e.g., `Panel`, across the respective class `Panel` and the aspect `ClickChain`. In the FOP implementation the code associated to one handler is scattered across the feature modules BASE and CLICKCHAIN.

In aggregation we balance the cohesion of the pattern implementations to be equivalent regarding the cohesion.

Variability. To evaluate variability, we have to consider two issues:

Varying the classes: In the OOP implementation the handler classes, e.g., `Frame`, `Panel`, and `Button`, have to implement the interface `ClickHandler`.
In the AOP implementation every class can be assigned to be member of the chain, i.e. to handle a click event. The FOP implementation similarly allows to assign every class to be a click handler.

Varying the chain: In the OOP implementation the code regarding the different chain elements is loosely coupled and thus changes to the events invoking the chain, e.g., a click onto a button, or the event handling methods, e.g., `handleClick`, can be applied through subclassing, i.e., the set of events can be extended only.
In the AOP implementation the definition of variant triggering events (pointcut `eventTrigger`) is tangled with the code of the graphical elements of the chain, e.g., `handleRequest`, thus they can not be exchanged independently.
In the FOP implementation the triggering events can be exchanged without changing the event handling methods by replacing the feature module TRIGGEREVENTS (Fig. 32).

Criteria	OOP	AOP	FOP
Cohesion	0	0	0
Variability	0	0	+

Table 5. Evaluation of the pattern Chain of Responsibility

4.5.4 Summary

The curse of the dominant decomposition [27] causes code tangling in the OOP and AOP solutions. The FOP implementation either detaches the code regarding the chain from the handler objects, e.g., of type `Frame`, and the implementation separates the class extensions of the different handlers, i.e., two dimensions are separated.
In the AOP implementation Hannemann et al. used ITD to extend an interface hosted by the same aspect, we advice to use abstract classes for that.
The hashmap implementation enforces the programmer to implement a lot of typecasts. That is tedious and error prone [19].
The *reusable* classes left `Panel`, `Frame`, and `Panel` are *empty*, beside the hook method `doClick` of the `Button` class that also is empty. We argue that this design is bad.
A summary is given in Table 5.

4.6 The Command Design Pattern

4.6.1 Intention

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations [13].

4.6.2 Implementation

OOP solution. Hannemann et al. apply the pattern to assign different actions to a button element. The button refers to one action of type `Command` by a field `command` (33). The type of object to perform the command, e.g. `ButtonCommand` or `ButtonCommand2`, can be exchanged with respect to the interface `Command`. The method `clicked` is invoked by the action listener and forwards requests to the member `command`.

Advantages Actions can be manipulated, extended, stored, and made undone by manipulating the `Command` objects. The tracing of different method invocations can be achieved by logging the methods of a command class. The actions, e.g., `ButtonCommand1`, applied to the button object can vary at runtime.
One command can be refered to by many classes thus improving reuse.

Disadvantages If objects of a class, e.g., `ButtonCommand1`, shall be invoked to perform actions, these classes have to implement the interface `Command`. The forwarding method `clicked` decreases performance. The code regarding the variant action performing objects is tangled with the main concern of the class `Button` by the field and the method `clicked`.
The `Command` object, e.g., of type `ButtonCommand1`, can not use private or protected members to perfórm actions after a click. The code associated to the `Button` class is scattered accross the classes `Button`, `Command`, `ButtonCommand` and `ButtonCommand2`.

AOP solution. The concrete action to invoke for each `Button` object is assigned inside a hashmap member of the aspect `CommandProtocol` (Fig. 34). The aspect methods `setCommand` and `get-`

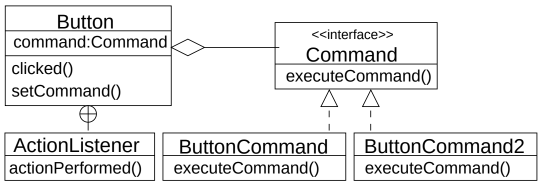

Figure 33. OOP implementation of the Command pattern.

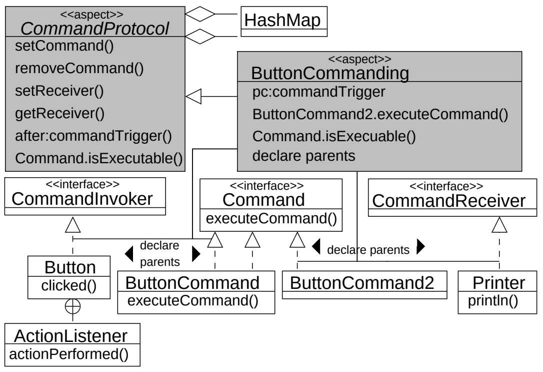

Figure 34. AOP implementation of the Command pattern.

`Command` manipulate the action objects, e.g., of type `ButtonCommand1`, assigned to buttons. The action regarding one button, i.e., the assigned hashmap element, is invoked by advice that extends the hook method of the `Button` class (`clicked`) that is called when a `Button` object is clicked.
Additionally, the AOP implementation assigns parameter objects, e.g., of type `Printer`, to each action object, e.g., `ButtonCommand1`. These action parameters are assigned in an additional hashmap for each `Button` object. The action parameter is assigned by the classes using the aspect methods `getReceiver` and `getReceiver`. The aspect `ButtonCommanding` introduces the commands to invoke after an event, e.g., `executeCommand`, into the `ButtonCommand2` class.

Advantages The advantages of the OOP implementation hold for the AOP implementation. Additionally, the `Button` class is not aware that its objects forward method calls to `Command` objects.

Disadvantages The code of the method `clicked` is scattered across the respective class `ButtonCommand2` and the aspect `ButtonCommanding`.
The hashmap implementation demands for multiple type casts due to empty interfaces.
In this pattern multiple inheritance is introduced. The aspect `ButtonCommanding` implements the method `isExecutable` of the interface `Command` using ITD. Due to parent declarations the method `isExecutable` of the interface `Command` is inherited twice.
The usage of a hashmap introduces performance drawbacks due to indirect method calls (Two indirect method calls are needed to compute the command object associated to a button-click and to compute the parameter of the action object.)
Hook methods, e.g., `Button.clicked`, have to be anticipated and complicate the code but are necessary to invoke the action performing classes, e.g., `ButtonCommand` and `ButtonCommand2`.

Criteria	OOP	AOP	FOP
Cohesion	0	+	+
Variability	0	+	+

Table 6. Evaluation of the pattern Command

FOP solution. We present 2 FOP solutions: Our solution A is close to the AOP implementation (Fig. 35). Similar to the AOP implementation two hashmaps are used to assign the button to an action performing class, e.g., `ButtonCommand` or `ButtonCommand2`, and to assign the parameter of the command, e.g., a `Printer` object. The hashmap is kept in a singleton object. The singleton object is invoked by the method extension `ButtonCommanding.ActionListener.actionPerformed` to get the associated action object. In solution B we omit the empty class `ButtonCommand2` in the feature module PREBASE and BASE but introduce the fully defined class in the feature module BUTTONCOMMANDING.

Advantages The advantages of the OOP and of the AOP implementation hold for the FOP implmentation.

Disadvantages Using hashmaps causes several type casts. Solution A scatters the feature module that implements the commanding into PREBASE and COMMANDING to avoid multiple inheritance.

4.6.3 Discussion

Cohesion. In the OOP implementation the class `ButtonCommand2` is closely coupled to variant implementations of the `clicked` method of the `Command` interface. In the AOP and FOP implementations the class `Buttoncommand2` is decoupled from the variant `Command` interface.

Variability. We have to consider 2 issues:

- In the OOP implementation classes that objects are assigned to perform actions after a click on a button, e.g., `ButtonCommand` or `ButtonCommand2`, are restricted to be a subclass the `Command` interface and to provide the method `executeCommand`. In the AOP and FOP implementation every class can be used to perform actions after a click event. The subtype declaration regarding the `Command` interface and the `executeCommand` method are introduced subsequently.
- In the OOP implementation the `Button` objects depend on objects of the classes `Command`, `ButtonCommand` and `ButtonCommand2`. In the AOP as in the FOP implementation the classes `Button`, `ButtonCommand`, and `ButtonCommand2` can be used with or without each other, they can vary flexible.

4.6.4 Summary

The reusable classes `ButtonCommand2` of the AOP and FOP implementations are empty despite an empty hook method thus the benefit in reuse is questionable.

4.7 The Composite Design Pattern

4.7.1 Intention

Compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly [13].

4.7.2 Implementation

OOP solution. The pattern is used to model a file-system tree (FST) including atomic `File` objects and composed `Directory` objects. The `Directory` and `File` objects can vary with respect to the interface `FileSystemComponent` (Fig. 37) and thus one

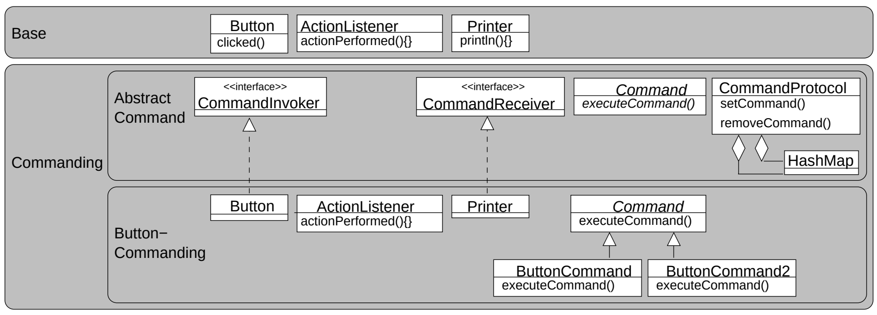

Figure 35. FOP implementation of the Command pattern.

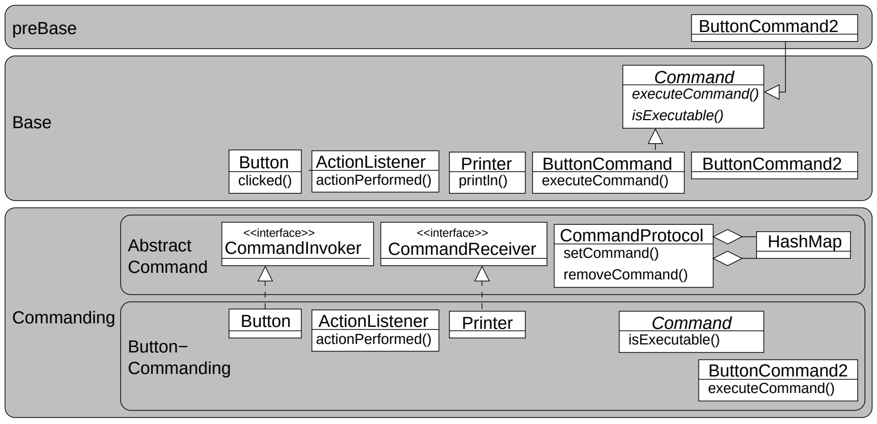

Figure 36. Alternative FOP implementation of the Command pattern.

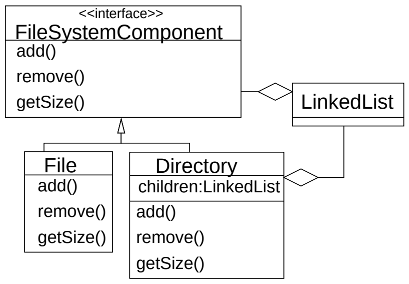

Figure 37. OOP implementation of the Composite pattern.

compound directory object can refer to files or nested directory objects.

Advantages Composed and atomic objects, i.e., `Directory` and `File` objects, can be exchanged without affecting the manipulating class, e.g., the `main` method.

Disadvantages The methods manipulating the FST are scattered across the interface `FileSystemComponent` and its subclasses `File` and `Directory`. Equivalently, functions to be applicable for the whole FST, e.g., `getSize`, are scattered. Composite classes, e.g., `File` or `Directory`, depend on the interface `FileSystemComponent` to be reusable.

AOP solution. In the AOP implementation the aspect `CompositeProtocol` merges the methods to compose the FST classes, e.g., `CompositeProtocol.add`, see Figure 38. That manipulates a hashmap of a singleton object. Hence, the aspect is used as a singleton. The aspect `FileSystemComposition` introduces the recursive functions, `subSum` and `printStructure` to be applied on the FST.

Advantages The advantages of the OOP implementation hold for the AOP implementation.
Methods to be applied onto the whole FST, e.g., computing the sum of the tree elements (`subSum`), are merged into the aspect `FileSystemComposition`.

Disadvantages Recursive methods, e.g., `printStructure` or `subSum` to be performed on the FST, are scattered across the aspects `CompositeProtocol` and `FileSystemComposition` (the method `FileSystemComponent.printStructure` that is defined and uses the methods `recurseOperation` and `recurseFunction` of the aspect `CompositeProtocol` (Fig. 39, Lines 13–18) and give over an anonymous visitor class (Line 14). Thus, the AOP implementation appears very complex compared to the OOP implementation that is depicted in Figure 40.
The implementation of the recursive functions `printStructure` (Fig. 39) and `subSum` introduces several indirections into the control flow at runtime due to the hashmap evaluation and the application of the visitor object.
The implementation of the method `subSum` for the class `File` is tangled with the implementation of the method `subSum` for other classes. This also holds for the `printStructure` method. Furthermore, the definitions of the different methods `subSum` and `printStructure` are coupled with each other in the aspect.

FOP solution. We present 2 FOP approaches for that pattern: Solution A is close to the AOP implementation, see Figure 41. The

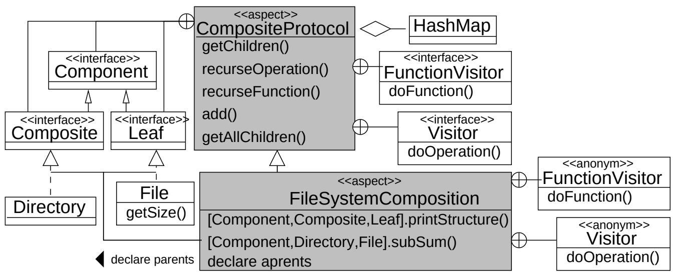

Figure 38. AOP implementation of the Composite pattern.

```
CompositeProtocol{
 public Enumeration recurseFunction(Component c,
   FunctionVisitor fv) {
  Vector results = new Vector();
  for (Enumeration enumM = getAllChildren(c); enumM.
    hasMoreElements(); ) {// method calls to their
    children
   Component child = (Component) enumM.nextElement();
   results.add(fv.doFunction(child));
  }
  return results.elements();
 }
}

FileSystemComposition extends CompositeProtocol{
 public int Directory.subSum() {
  Enumeration enumM = FileSystemComposition.
    getInstance().recurseFunction(this , new
    FunctionVisitor() {
   public Object doFunction(Component c) {
    return new Integer(c.subSum());
   }
  });

  int sum = 0;
  while (enumM.hasMoreElements()) {
   sum += ((Integer) enumM.nextElement()).intValue();
  }
  return sum;
 }
}
```

Figure 39. AOP recursive function using an anonymous visitor class and indirection to aspect.

```
private static void
  printStructure(FileSystemComponent comp) {
 indent();
 System.out.println(comp);
 indent +=4;
 for (int i=0; i<comp.getChildCount(); i++) {
  printStructure(comp.getChild(i));
 }
 indent -= 4;
}
```

Figure 40. External OOP implementation traversing an recursive structure.

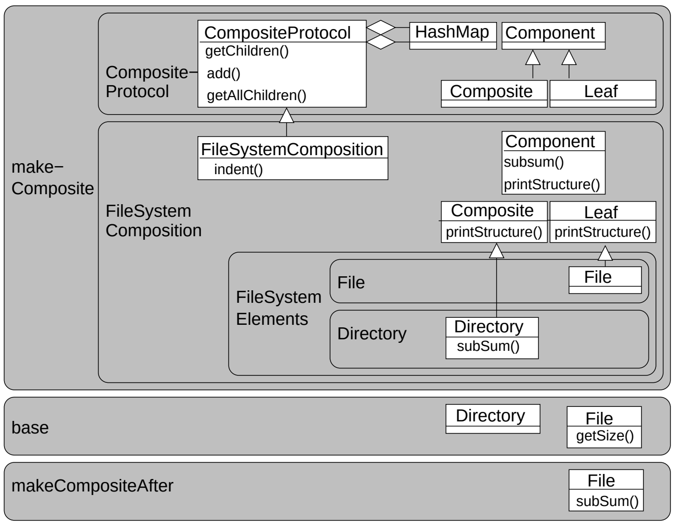

Figure 42. Alternative FOP implementation of the Composite pattern.

FST-classes are assigned to the FST object using a hashmap in the singleton class `FileSystemComposition`. The recursive functions are assigned to the FST-classes using mixins for each FST-class. The visitor classes `Visitor` and `FunctionVisitor` are used by the recursive methods.
In solution B (Fig. 42) the recursive functions are assigned to each class of the FST according to the type of the FST-class. We omitted the visitor by invoking the children retrieved from the singleton class `FileSystemComposition` directly (Fig. 43). The children are retrieved (Line 5), traversed (Line 6), and directly invoked (Line 7). Additionally, the refinements that compose FST classes (e.g., feature module FILESYSTEMCOMPOSITION) are separated from the mixin classes of the FST classes (feature module FILESYSTEMELEMENTS).

Advantages The advantages of OOP hold for the FOP implementations.
The code to compose the FST is merged into the singleton class `CompositeProtocol`. The recursive functions `printStructure` and `subSum` to be applicable for the FST are not scattered but merged in the feature modules FILESYSTEMCOMPOSITION and MAKECOMPOSITEAFTER.
Solution B only introduces one indirection for retrieving the child of one FST element out of a hashmap to compute a recursive function. Figure 43 shows, that the child elements of one FST element are gathered from the singleton `FileSystemComposition` (Lines 5–6) and are accessed directly (Line 7).

Disadvantages We used mixin layers to implement FOP, that enforced us to split the extension of the `File` class so that it can access members that are introduced in the class `Base.File`.

4.7.3 Discussion

Cohesion. We have to consider three issues for evaluating cohesion of these implementations:

- OOP scatters the implementation of the recursive functions, e.g., `getSize`, across the classes `FileSystemComponent`, `File`, and `Directory`. In the AOP implementation the implementation of each recursive function is scattered, e.g., `subSum` or `printStructure`, across the aspects `CompositeProtocol`,

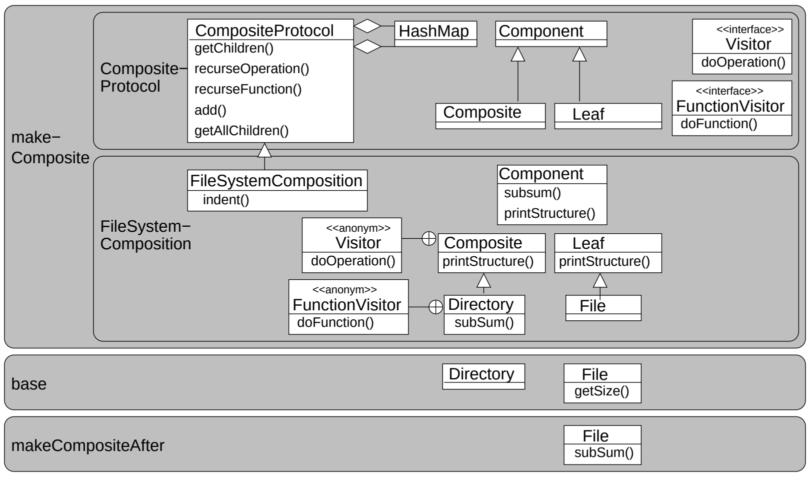

Figure 41. Direct FOP implementation of the Composite pattern.

```
public void printStructure(final PrintStream s) {
 FileSystemComposition.indent();
 s.println("<Composite>"+this);
 FileSystemComposition.indent +=4;
 for (Enumeration _enum = FileSystemComposition.
   getInstance().getAllChildren(this); _enum.
   hasMoreElements(); ) {
  Component child = (Component) _enum.nextElement();
  child.printStructure(s);
 }
 FileSystemComposition.indent -=4;
}
```

Figure 43. Internal FOP implementation traversing an recursive structure without anonymous visitor.

and `FileSystemComposition`. FOP only scatters the code of recursive functions, that have to access members of the base class, e.g., `subSum` across the feature modules MAKECOMPOSITE and MAKECOMPOSITEAFTER. Other recursive functions, like `printStructure` can be implemented cohesively in the feature module MAKECOMPOSITE.

- The OOP implementation merges the code associated to each FST element, i.e., `File` and `Directory`, in each respective class. In the AOP implementation the code regarding each FST element class, e.g., `File`, is scattered across the aspects `CompositeProtocol`, and `FileSystemComposition`. In the FOP implementation the code regarding the file system components is scattered across the feature modules MAKECOMPOSITE and MAKECOMPOSITEAFTER.
- In the OOP implementation the code to compose the FST, e.g., the methods `add` or `remove`, is scattered across all classes and interfaces. The AOP and FOP implementation merges the code to compose the FST in the aspect `CompositeProtocol` and the feature module COMPOSITEPROTOCOL respectively.

Variability. We have to consider 2 issues:

- In the OOP implementation the classes to act as file system components are restricted to those implementing the `FileSystemComponent` interface. In the AOP and FOP implementations every class can be assigned to be part of the FST.
- In the OOP approach the implementation of single FST classes can vary independently. The AOP implementation tangles the code of different FST classes and thus prevents variation. The FOP solution (B) provides to exchange implementations of single classes of the FST.

Criteria	OOP	AOP	FOP
Cohesion	0	0	0
Variability	0	0	+

Table 7. Evaluation of the pattern Composite

4.7.4 Summary

The code of Figure 39 and Figure 40 prints the composite structure using AOP and OOP mechanisms respectively. We argue, that the method of Figure 40 is much more convinient to the programmer than the AOP implementation (Fig. 39) that introduces multiple indirections, anonymous classes and aspects.
A summary is given in Table 7.

4.8 The Decorator Design Pattern

4.8.1 Intention

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclasing for extending functionality [13].

4.8.2 Implementation

OOP solution. Hannemann et al. applied the pattern to enhance the output of a printer object (`ConcreteOutput`) by introducing additional characters into the printed string object. Decoration objects (e.g., of type `StarDecorator` and `BracketDecorator`) are used instead of the printer objects (of type `ConcreteOutput`). The decorator objects refer to the printer object by an object reference and forward print requests to the referred printer object. Before or after forwarding the requests the parameter object of type `String` is enhanced by additional characters, e.g., brackets (in class `BracketDecorator`) or stars (in class `StarDecorator`). To replace the printer objects, the decorator objects, e.g., of type `BracketDecorator`, have to implement the same interface `Output`, see Figure 44.

Advantages Decorator objects can be exchanged at runtime.
The printer class `OutputImplementation` can be exchanged without need for code replication or inheritance.

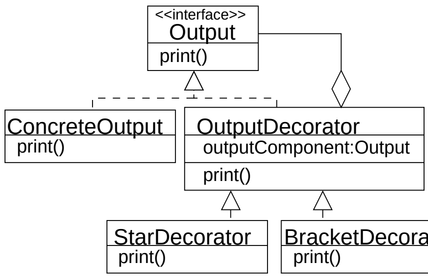

Figure 44. OOP implementation of the Decorator pattern.

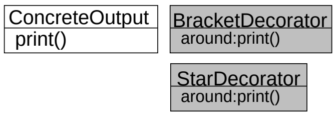

Figure 45. AOP implementation of the Decorator pattern.

Disadvantages Decorators, e.g., `BracketDecorator`, include forwarding methods *for all* methods declared in the printer interface, e.g., `print`, thus decreasing performance.
Decorator objects only can extend public members of the printer objects.
To decorate a method using the design pattern Decorator at least this method has to be bound dynamically.[9] In the implementation presented all methods of the printer class `ConcreteOutput`, e.g., `print`, are bound dynamically thus decreasing performance and improving resource consumption [10].

AOP solution. To enhance the argument string of the printer with additional characters, the aspects `StarDecorator` and `BracketDecorator` intercept the calls to the printer functions using PCA. The argument string of each method call is gathered from the PC and decorated with additional parameters. The original `print` method of the class `ConcreteOutput` is invoked with the extended parameter object. The decoration is applied or omitted based on properties of the control flow.

Advantages Since single methods, like `print`, are decorated, non decorated methods of the `ConcreteOutput` class stay unaffected.
Clients of the class `ConcreteOutput`, e.g., the `main` method, are not affected, if a decoration is applied.
No virtual methods are necessary, i.e., performance and resource consumption are improved.

Disadvantages We did not observe specific disadvantages for that implementation.

FOP solution. In our FOP implentation we extend the method `ConcreteOutput.print` using method extensions. This method extension augments the parameter string to print.

Advantages The advantages of the AOP implementation hold for the FOP implementation.

Disadvantages The decoration is applied statically, i.e., every call to the method invokes the decoration unaffected by the calling object or the dynamic control flow.

[9] If the decorators would be subclasses of the decorated printer class (not depicted).

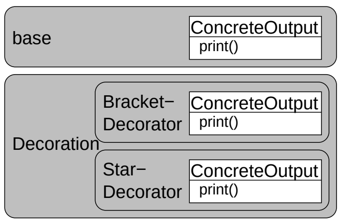

Figure 46. FOP implementation of the Decorator pattern.

Criteria	OOP	AOP	FOP
Cohesion	0	0	+
Variability	0	+	+

Table 8. Evaluation of the pattern Decorator

4.8.3 Discussion

Cohesion. In the OOP implementation the decorating code is scattered across the classes `OutputDecorator`, `StarDecorator`, and `BracketDecorator`. In the AOP implementation the decoration code is scattered across the aspects. In the FOP implementation the decoration code is merged into the feature module DECORATION.

Variability. In the OOP implementation classes that should be decorated are restricted to implement an interface, e.g., `Output`, or to bind its methods dynamically. The AOP and FOP approaches allow to decorate every method of every class although this method might not be bound dynamically.

4.8.4 Summary

In the OOP implementation the decorating class has to implement every method of the decorated object thus methods that are not augmented with statements but forwarded only decrease performance. In the AOP and FOP implementation no primitive forwarding methods are needed at all which improves the performance.

In the OOP implementation decorator objects can be exchanged at runtime. In AOP the decoration of methods can be applied based on dynamic properties of the control flow. FOP only provides the static application of decorations.

An overview is given in Table 8.

4.9 The Facade Design Pattern

4.9.1 Intention

> Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use [13].

4.9.2 Implementation

OOP solution. The pattern is applied to hide the subsystem of classes (`Decoration`, `RegularScreen`, and `StringTransformer`) that is used to perform different operations that transform a string. Instead, the facade object is invoked. The operation is implemented in the method `OutputFacade.printFancy`, which invokes the subsystem.

Advantages The interaction of a set of classes is modularized into the `OutputFacade` object and thus the complexity of the invoking classes decreases.

Disadvantages The pattern does not prevent direct access to the subsystem components.

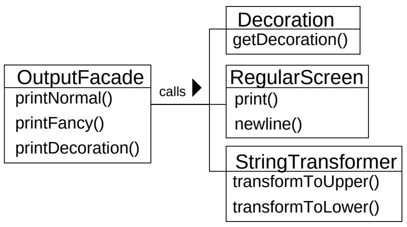

Figure 47. OOP implementation of the Facade pattern.

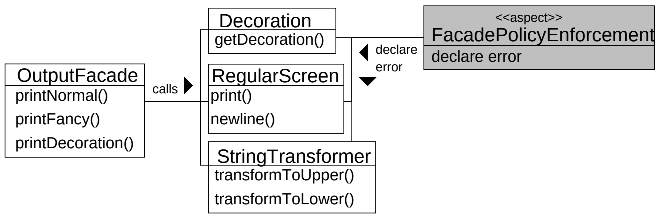

Figure 48. AOP implementation of the Facade pattern.

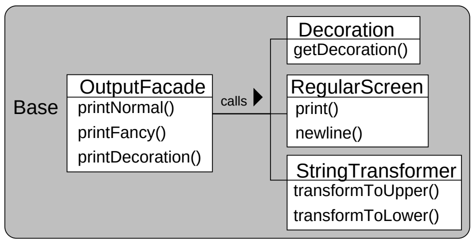

Figure 49. FOP implementation of the Facade pattern.

AOP solution. The aspect `FacadePolicyEnforcement` restricts the access to the subsystem that performs the string transformation. If classes of the subsystem are invoked by classes other than the `OutputFacade` class, a compiler warning is given.

Advantages The advantages of the OOP implementation hold for the AOP implementation. The compiler warning introduced by the aspect supports the development of loosely coupled classes but does not imply that.

Disadvantages We did not observe obvious disadvantages of the AOP implementation

FOP solution. We transfered the OOP implementation into a feature module. That is, the method `printFancy`, invokes the subsystem classes `Decoration`, `RegularScreen`, and `StringTransformer` to perform an operation.

Advantages The advantages of the OOP implementation hold for the FOP implementation.

Disadvantages The pattern does not prevent the client classes which use the subsystem to directly invoke methods of the subsystem classes.

4.9.3 Discussion

Cohesion. The OOP, AOP, and FOP implementations are equivalent with respect to cohesion.

Criteria	OOP	AOP	FOP
Cohesion	0	0	0
Variability	0	0	0

Table 9. Evaluation of the pattern Facade

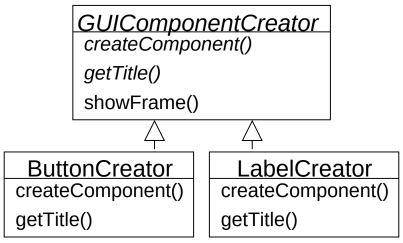

Figure 50. OOP implementation of the Factory Method pattern.

Variability. The OOP, AOP, and FOP implementations are equivalent regarding the variability.[10]

4.9.4 Summary

A summary is given in Table 9.

4.10 The Factory Method Design Pattern

4.10.1 Intention

> Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses [13].

4.10.2 Implementation

OOP solution. The pattern is applied to create a frame that includes a variant graphical object, e.g., of type `JButton` or `JLabel`. The type of the element is determined by the methods `getTitle` and `createComponent`, i.e., these methods create the graphical object that is put on the frame. These object creating methods are used by the method `GUIComponentCreator.showFrame` that defines the abstract algorithm. The implementations of these methods can vary depending on the instantiated class, e.g., `ButtonCreator`. Subsequently we refer to the method `showframe` as *abstract algorithm method*. The methods creating the graphical objects are called *factory methods*.

Advantages An algorithm can be applied to elements of a variety of types.

Disadvantages Clients of the factory method class `GUIComponentCreator` are closely coupled to the implementation of the factory methods because the clients select the implementations by instantiating the respective subclass of the class `GUICompoenentCreator`. The different graphical elements to be put onto the frame have to implement the common interface `JComponent`.
If the `showFrame` method should be exchanged or a new abstract algorithm method should be introduced, either the class `GUIComponentCreator` has to be replicated or changed invasively. Both demand for replication of the subclasses `ButtonCreator` and `LabelCreator`.

[10] The AOP compiler warning supports the developement of loosely coupled classes, which can be composed variable. Nevertheless the aspect does not introduce additional flexibility.

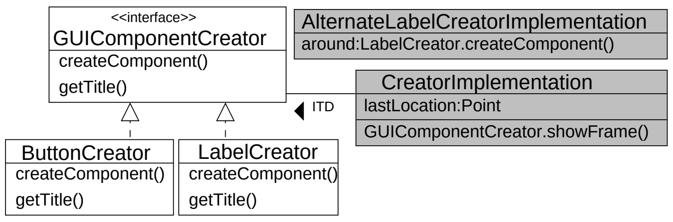

Figure 51. AOP implementation of the Factory Method pattern.

AOP solution. In the AOP implementation the variant abstract algorithm method `showFrame` is transfered into the aspect `CreatorImplementation` that introduces it on demand. The aspect `AlternateLabelCreationImplementation` overrides the factory method `createComponent` of the `LabelCreator` class.

Advantages The advantages of the OOP implementation hold for the AOP implementation. If the abstract algorithm method `showFrame` should be exchanged, no subclasses have to be replicated since the inherited class is not exchanged.

Disadvantages Modules implementing different variants of the software are not separated from modules implementing essential issues of the implementation.

FOP solution. We present two implementation approaches for that pattern: solution A is close to the AOP implementation, see Figure 52. We detach the variant abstract algorithm method `showFrame` into the feature module CREATORIMPLEMENTATION. The method extension `AlternateLabelCreatorImplementation.LabelCreator.createComponent` overrides the method `Base.LabelCreator.createComponent`.
Solution B (Fig. 53) allows to compose the abstract algorithm methods with the template methods statically. That is, the graphical elements of every frame may vary at compile time but are invariant at runtime.

Advantages The advantages of the OOP and AOP implementations hold for the FOP implementation.

Solution A modularize variant extensions of the essential code cohesively in the feature module FACTORYEXTENSION.

Solution B reduces the number of methods that are dynamically bound because there are no subclasses. This improves performance and resource consumption.

Disadvantages We did not observe disadvantages of the FOP implementation A. In solution B the different factory methods, e.g., `createComponent`, can not vary at runtime.

4.10.3 Discussion

Cohesion. In the OOP implementation variant implementations of the `showFrame` method are coupled with the essential `GUIComponentCreator` method declarations (e.g., `createComponent` and `getTitle`). The AOP implementation detaches variant extensions of the essential `GUIComponentCreator` methods but scatters different extensions across the aspects `CreatorImplementation` and `AlternateLabelImplementation`. In the FOP implementation the variant extensions of the `GUIComponentCreator` class are merged in the feature module FACTORYEXTENSION.

Variability. In the OOP implementation the variation of the abstract algorithm method `showFrame` causes either code replication or invasive changes. In the AOP implementation the abstract algorithm method can be exchanged flexible by exchanging the aspect, e.g., `CreatorImplementation`, to be applied – code replication is prevented. The FOP implementation allows to exchange the abstract algorithm method without code replication by exchanging the mixin classes, i.e., the feature modules.

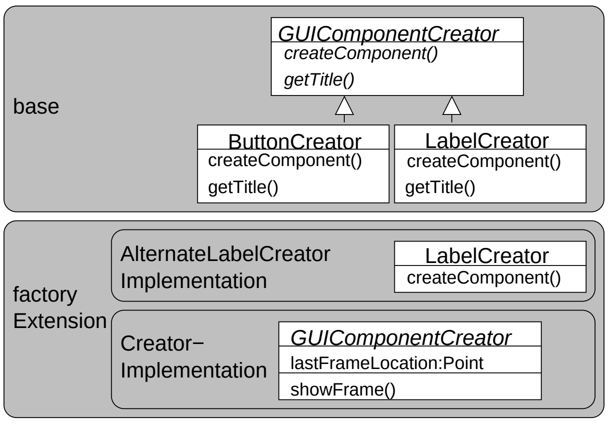

Figure 52. FOP implementation of the Factory Method pattern.

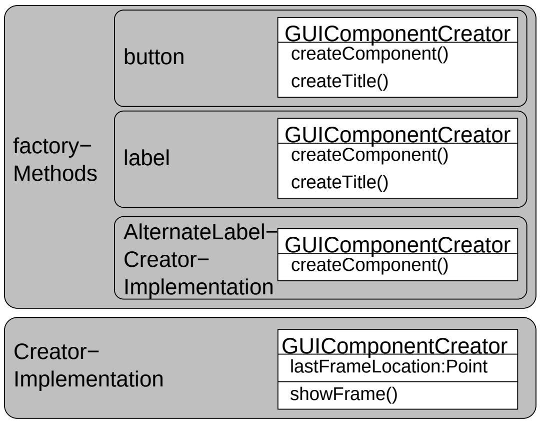

Figure 53. Alternative FOP implementation of the Factory Method pattern.

Criteria	OOP	AOP	FOP
Cohesion	-	0	+
Variability	0	+	+

Table 10. Evaluation of the pattern Factory Method

4.10.4 Summary

To override methods of final classes we advice the usage of OOP inheritance if applicable instead of replacing the method using `around` advice.

4.11 The Flyweight Design Pattern

4.11.1 Intention

Using sharing to support large numbers of fine-grained objects efficiently [13].

4.11.2 Implementation

OOP solution. Hannemann et al. applied the pattern to store a set of characters. To improve the resource consumption they only store the extracted property of the type of characters and omit the property of capitalization. Furthermore, whitespaces are stored by

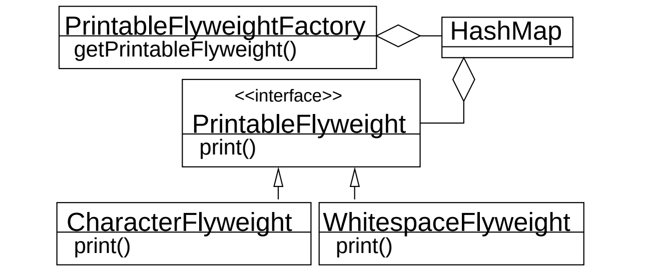

Figure 54. OOP implementation of the Flyweight pattern.

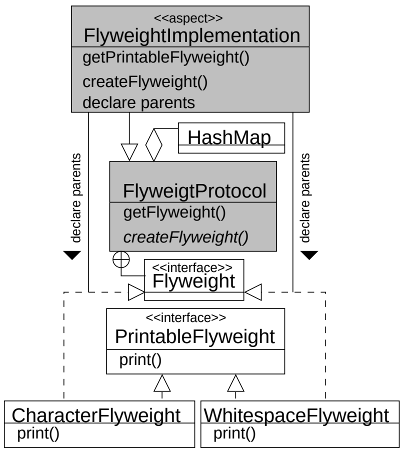

Figure 55. AOP implementation of the Flyweight pattern.

an additional class.
A pool of character objects (of type `CharacterFlyweight`, Fig. 56) and whitespace objects (of type `WhitespaceFlyweight`) that are shared is stored in a hashmap of the class `PrintableFlyweightFactory`. To reuse the character objects, the omitted property of capitalization has to be defined as parameter for the object.

Advantages The resource consumption of multiple character objects can be reduced by reusing one shared state variable.

Disadvantages Flyweights may introduce performance penalties due to hashmap evaluation and parameter evaluation.

AOP solution. The AOP implementation is similar to the OOP implementation but manages the flyweight pool, i.e., the hashmap, inside the aspects `FlyweightProtocol` and `FlyweightImplementation`. The AOP implementation attend the difference between the usage of `Flyweight` objects that are stored in the hashmap and of `PrintableFlyweight` objects that are returned to a client The flyweight objects are assigned to the interface by the aspect. The character using method `main` uses flyweight objects by manipulating the `FlyweightImplementation` aspect which acts as a singleton.

Advantages The advantages of the OOP implementation hold for the AOP implementation.

Disadvantages The empty interface introduced by the aspect enforces the implementation of type casts.
The implementation may introduce performance penalties due to hashmap and parameter evaluation.

FOP solution. We present two FOP solutions: solution A, that is depicted in Figure 56, is close to the AOP implementation. The

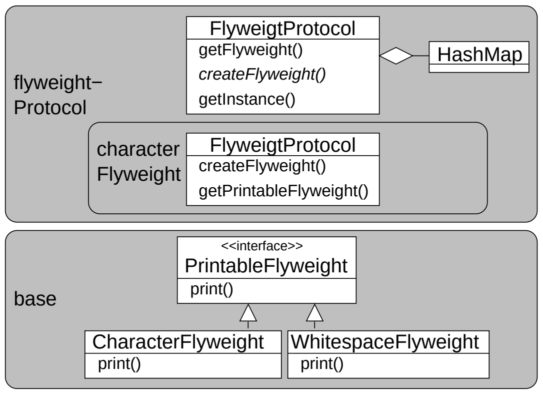

Figure 57. Simplified FOP implementation of the Flyweight pattern.

Criteria	OOP	AOP	FOP
Cohesion	+	0	+
Variability	0	+	+

Table 11. Evaluation of the pattern Flyweight

shared character flyweight objects are stored in a hashmap object of the singleton class `Flyweightprotocol`. That singleton class is manipulated by the clients, e.g., the `main` method, to retrieve flyweight objects.
Solution B reduces the number of classes and feature modules by omitting the `Flyweight` interface, see Figure 57. Hence, the hashmap stores flyweight objects using the static type object (as it is done anyway in the hashmap). Type casts are applied to turn these objects (of type `Object`) into `Flyweight` objects. (The type casts has been applied in solution A too, thus *no additional* type casts are introduced.)

Advantages The advantages of the OOP and AOP implementations hold for the FOP implemenation.

Disadvantages In solution A the empty interface `Flyweight` introduced by the feature module FLYWEIGHTPROTOCOL demands for type casts, that are error prone.

4.11.3 Discussion

Cohesion. In the OOP implementation the creation of `PrintableFlyweight` objects is merged in one class. In the AOP implementation the creation of `PrintableFlyweight` objects is scattered across the aspects. The FOP implementation merges the code creating `PrintableFlyweight` objects in the feature module FLYWEIGHTPROTOCOL.

Variability. In the OOP implementation the classes to be used as flyweights are restricted to those that are subtype of the interface `PrintableFlyweight`. In the AOP and FOP implementations every class can be used to create flyweight objects since the subtype declaration regarding a uniform interface `Flyweight` is introduced subsequently (using `declare parents` statements in AOP and mixins in FOP).

4.11.4 Summary

A summary is given in Table 11.

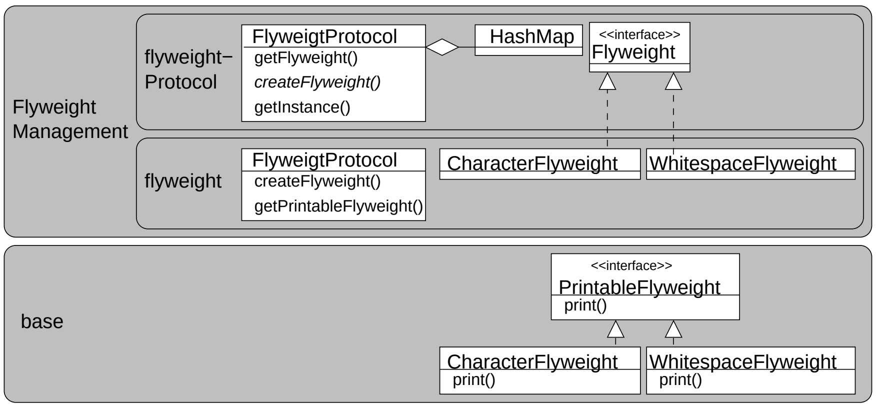

Figure 56. FOP implementation of the Flyweight pattern.

4.12 The Interpreter Design Pattern

4.12.1 Intention

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language [13].

4.12.2 Implementation

OOP solution. The pattern is applied to evaluate boolean expressions (BE). Each operator of the language, e.g., *AND*, *OR*, and *NOT*, is implemented by a class, e.g., `AndExpression`, `OrExpression`, and `NotExpression`, see Figure 58. These operator classes are used to compose expressions of boolean variables and constants. The composed BE are evaluated by assigning different values to the boolean variables. The boolean constants ”true” or ”false” are coded in the class `BooleanConstant`; the boolean variables are implemented by the class `VariableExpression`. The classes of the boolean operators, e.g., `AndExpression`, and the boolean constant and variable classes (`BooleanConstant` and `VariableExpression`) can vary with respect to the common interface `BooleanExpression`. Consequently, a boolean operator can compose boolean constants, variables or nested boolean expressions including further operators, i.e., nested and complex BE can be composed. The assignement of values to the boolean variables is done using the singleton class `VariableContext` and its hasmap field respectively.

Advantages The language of boolean expressions is easy to extend, e.g., a new operator *XOR* could be implemented in a subclass of the `BooleanExpression` interface.

Disadvantages Recursive methods, e.g., `evaluate`, to perform on BE are scattered across the interface `BooleanExpression` and all subclasses, e.g., `AndExpression`. Classes that should be used to build and analyze boolean expressions, e.g., `AndExpression`, are restricted to those providing the methods `evaluate`, `replace`, and `copy` and that are subtype of the interface `BooleanExpression`.

AOP solution. In the AOP implementation recursive methods to be applied on the BE, e.g., `evaluate`, are merged in the aspect `BooleanInterpretation`. The aspect distributes the methods to the BE-classes using ITD.

Advantages The advantages of the OOP implementation hold for the AOP implementation.

Disadvantages The different ITD of the BE methods, e.g., `evaluate`, are tangled within the aspect, i.e., altering the ITD of one class (e.g., `AndExpression.evaluate`) causes code replication for the ITD of the other classes (e.g., `OrExpression.evaluate`).
The implementation does not work without the aspect.

FOP solution. Our FOP implementation is close to the AOP implementation (Fig. 60). The recursive methods, e.g., `evaluate`, of the `BooleanExpression` subclasses are transfered into mixin classes which are merged in the feature module INTERPRETER.

Advantages The advantages of the OOP and AOP implementation hold for the FOP implementation.

Disadvantages The mixin of one method, e.g., `interpreter.AndExpression.evaluate`, can not vary with respect to other mixins, e.g., `interpreter.OrExpression.evaluate`. The feature module BASE does not work without the feature module INTERPRETER.

4.12.3 Discussion

Cohesion. To evaluate the cohesion of the different implementations, we have to consider two issues:

- OOP scatters the implementation of recursive methods to be applied on the BE, e.g., the evaluation of boolean expressions (`evaluate`), across different classes, e.g., `AndExpression`, and couples the code within these classes to the BE specific code. The AOP implementation merges the recursive methods of the BE of different classes into the aspect `BooleanInterpretation` but does not separate the variant modules from the modules that are not variant. In the FOP implementation the recursive methods are separated from the code specific to each operation into mixin classes, e.g., `interpreter.AndExpression`, and all mixin classes are merged in the feature module INTERPRETER.
- The OOP implementation merges the methods associated to one boolean operator, boolean variable and boolean constant respectively in classes. The AOP implementation scatters the recursive methods regarding one boolean operator, e.g., *AND*, across the respective classes, e.g., `AndExpression`, and the aspect `BooleanImplementation`. The FOP implementation as the AOP implementation scatters the code implementing one boolean operator, e.g., *AND*, across the respective class, e.g., `AndExpression`, and the feature module INTERPRETER.

Variability. We have to consider two issues:

- In the OOP implementation of recursive methods, like `evaluate`, can vary with respect to the operator classes, the `Varia-`

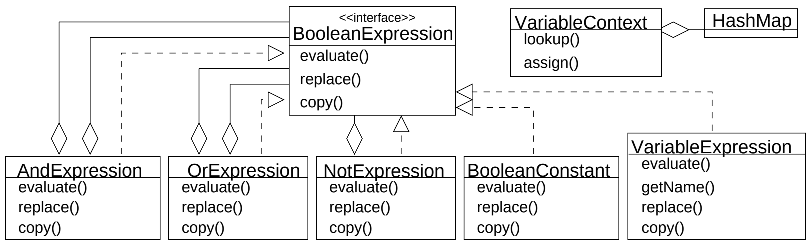

Figure 58. OOP implementation of the Interpreter pattern.

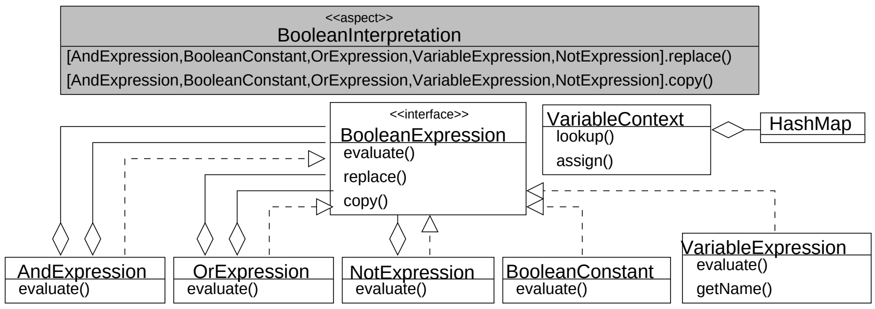

Figure 59. AOP implementation of the Interpreter pattern.

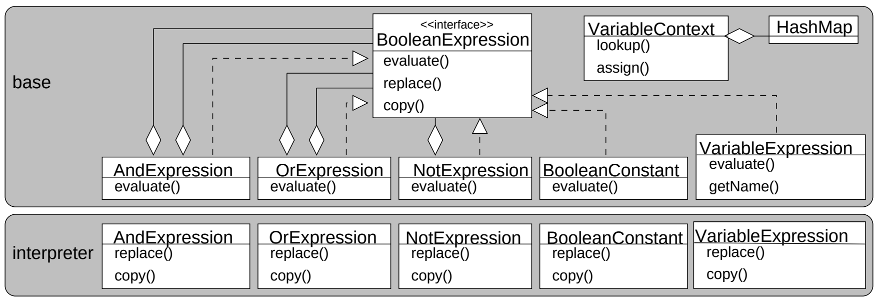

Figure 60. FOP implementation of the Interpreter pattern.

`bleExpression` class, and the `BooleanConstant` class. In the AOP and FOP implementation the recursive methods, i.e., the ITD and mixin associated to these methods, like `evaluate`, can be exchanged by exchanging the applied aspect (`BooleanInterpretation`) and feature module (INTERPRETER) respectively.

- In the OOP implementation the implementations of recursive methods, e.g., `evaluate`, can vary for each class. In the AOP and FOP implementation, the method specific ITD and mixins are tangled within the aspect `BooleanInterpretation` and the feature module INTERPRETER respectively and thus they can not be exchanged.

In summary the implementations are equivalent regarding the variability.

4.12.4 Summary

A summary is given in Table 12.

Criteria	OOP	AOP	FOP
Cohesion	0	0	+
Variability	0	0	0

Table 12. Evaluation of the pattern Interpreter

4.13 The Iterator Design Pattern

4.13.1 Intention

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation [13].

4.13.2 Implementation

OOP solution. Hannemann et al. applied the pattern to traverse a list in different ways, e.g., forward or reverse traversation. The traversation strategies, e.g., reverse traversation through the list, are an *iterator* class, e.g., `ReverseIterator`. That is, the iterator traverses elements of `OpenList` list objects.

Advantages Different traversation strategies for a list object, e.g., forward and backward traversation, can be exchanged without affecting the list classes.

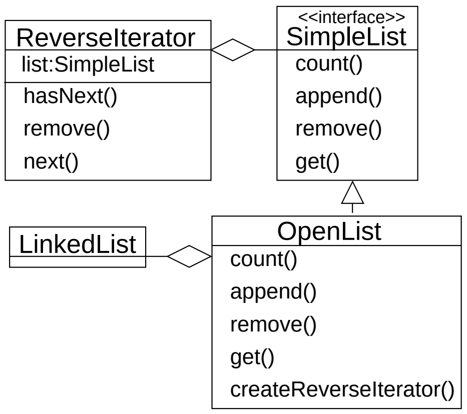

Figure 61. OOP implementation of the Iterator pattern.

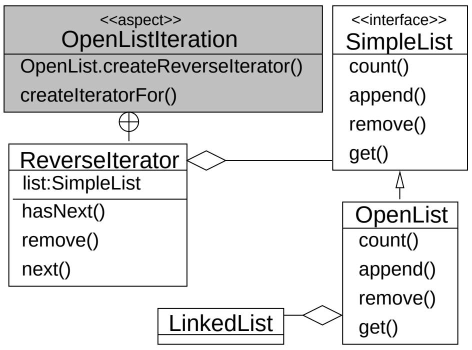

Figure 62. AOP implementation of the Iterator pattern.

Disadvantages The `ReverseIterator` object only can invoke public members of the class `OpenList` to taverse the `OpenList` objects; if this interface is restricted the possibilities for the iterator are restricted.
The `OpenList` class is closely coupled to the `ReverseIterator` class due to the return type of the method `OpenList.createReverseIterator`.

AOP solution. In the AOP implementation the method of the `OpenList` class that creates iterator objects, i.e., `OpenList.createReverseIterator`, is transfered into the aspect `OpenListIteration` (Fig.62). Furthermore, the iterator class (`ReverseIterator`) is encapsulated inside this aspect.

Advantages The advantages of the OOP implementation hold for the AOP implementation.
The `OpenList` class is decoupled from its iterator class `ReverseIterator` because the method `createReverseIterator` is separated from the list class.

Disadvantages There are no obvious disadvantages of the AOP implementation we observed.

FOP solution. Our FOP implementation is close to the AOP implementation, i.e., the method that creates the iterator objects for an `OpenList` object (`createReverseIterator`) is transfered into the mixin class `OpenList` of the feature module ITERATOR. The method `createReverseIteratorFor` provides an alternative way to create iterator objects for a list object.

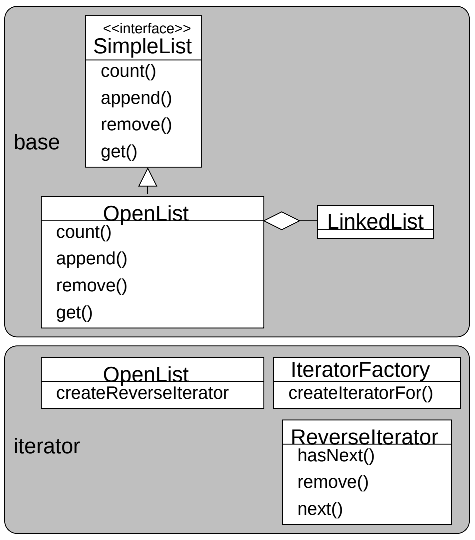

Figure 63. FOP implementation of the Iterator pattern.

Criteria	OOP	AOP	FOP
Cohesion	-	0	+
Variability	0	+	+

Table 13. Evaluation of the pattern Iterator

Advantages The advantages of the OOP and FOP implementations hold for the FOP implementation too.

Disadvantages The FOP implementation of the pattern depicts no obvious disadvantages.

4.13.3 Discussion

Cohesion. In the OOP implementation the `Openlist` class is closely coupled to the variant `ReverseIterator` class due to the return type of the `createReverseIterator` method.
In the AOP implementation the `OpenList` class is not coupled to the `ReverseIterator` class because the `createReverseIterator` method is introduced subsequently. But the AOP implementation does not separate variant aspects from essential classes.
In the FOP implementation the `OpenList` class is not coupled to the `ReverseIterator` class because the method `createReverseIterator` is introduced subsequently. The variant mixin classes of the feature module ITERATOR, e.g., `iterator.OpenList`, are separated from the essential classes (feature module BASE).

Variability. In the OOP implementation classes of iterator objects that can be created by the `OpenList` objects are restricted to be subtype of the `ReverseIterator` class due to the return-type of the method `createReverseIterator`. In the AOP and FOP implementations iterator objects of different types can be created by the `OpenList` objects.

4.13.4 Summary

A summary is given in Table 13.

4.14 The Mediator Design Pattern

4.14.1 Intention

Define an object that encapsulate how a set of objects interact. Mediators promotes loose coupling by keeping objects

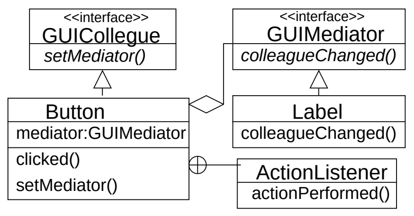

Figure 64. OOP implementation of the Mediator pattern.

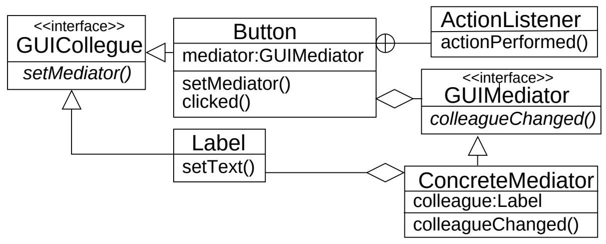

Figure 65. Mediator pattern by Gamma et al.

from referring to each other explicitly, and it lets you vary their interaction independently [13].

4.14.2 Implementation

OOP solution. In the OOP implementation a mediator class (`Label`, Fig. 64) is used by event originator objects (of type `Button`) to update associated colleague objects, e.g., of type `Label`. That is, after the `Button` object is clicked, the method `colleagueChanged` of the mediator object `Label` is invoked which updates the colleage object (i.e., it updates itself in this implementation of Hannemann et al.)[11]

Advantages The colleague classes, e.g., `Button` or `Label`, can vary with respect to the `GUIColleage` interface because the communication is implemented in the mediator class `Label`. That is, the colleague classes are decoupled and thus the `Button` object is not aware of the type of colleague it is updating. For that no other colleague, e.g., `Label`, class has to change.
The way objects interact is explicitely kept in the concrete mediator classes.

Disadvantages The mediator class itself gets complex and monolithic because the it merges the communication code between different types of objects.
In the implementation presented by Hannemann et al. the mediator code is tangled with the code of the colleague class `Label` thus preventing variation of the mediator and colleague.
The `Button` objects of class `Button` depend on a `GUIMediator` interface an object due to the member field that is declared and used in the `Button` class.

AOP solution. In the AOP implementation the update of colleage objects, e.g., of type `Label`, is merged into the aspect method `MediatorProtocol.notifyMediator`, see Figure 66. This method is invoked by advice after the button has been clicked. This method notifies the colleagues associated in a hashmap of the aspect `MediatorProtocol`.

[11] The implementation presented by Hannemann et al. differs from the approach of Gamma et al. [13]. The approach of Gamma et al. is depicted in Figure 65.

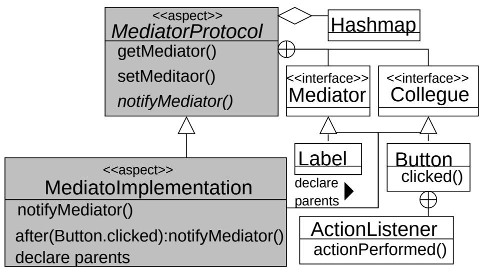

Figure 66. AOP implementation of the Mediator pattern.

Advantages The advantages of the OOP implementation hold for this AOP implementation. Classes to act as colleague, e.g., `Button`, do not have to be subtype of a specific interface `Colleague` but are extended subsequently to do so.
If the `Button` class should not invoke the mediator object this can be achieved by omitting the mediator aspect.

Disadvantages The update code is restricted for updating `Label` object only.

FOP solution. The FOP solution is close to the AOP implementation (Fig. 67).
Colleague objects are associated to `Button` objects in a hashmap of the singleton class `MediatorProtocol`. The notification of colleagues is applied by extending the `MyActionListener.actionPerformed` method. (FOP prevents the extension of anonymous classes thus we embodied the `ActionListener` class in the `MyActionListener` class.)

Advantages The FOP implementation overtakes the advantages of the OOP and FOP implementations.

Disadvantages The disadvantages of the AOP implementation hold for the FOP implementation.

4.14.3 Discussion

Cohesion. Two aspects have to be considered for the evaluation of cohesion:

- In Hannemanns implementation of the Mediator pattern the communication code in the mediator class `Label` is closely coupled to the colleague code of the class `Label`. In the AOP and FOP implementations variant communication code, e.g., `notifyMediator`, i.e, `colleagueChanged`, is separated from the communicating classes, e.g., `Button` and `Label`.
- In the OOP implementation the code for invoking the colleague is scattered across all classes. In the AOP implementation the objects code for notifying colleagues is scattered across the aspects. In the FOP implementation the code of notifying colleagues is merged into the feature module MEDIATOR.

Variability. In the OOP implementation the class `Button` depends on the classes `GUIMediator` and `Label` due fields and methods.
In the AOP and FOP implementations the `Button` class does not depend on the `GUIMediator` and `Label` classes because the mediator invocation is introduced subsequently.

4.14.4 Summary

A summary is given in Table 14.

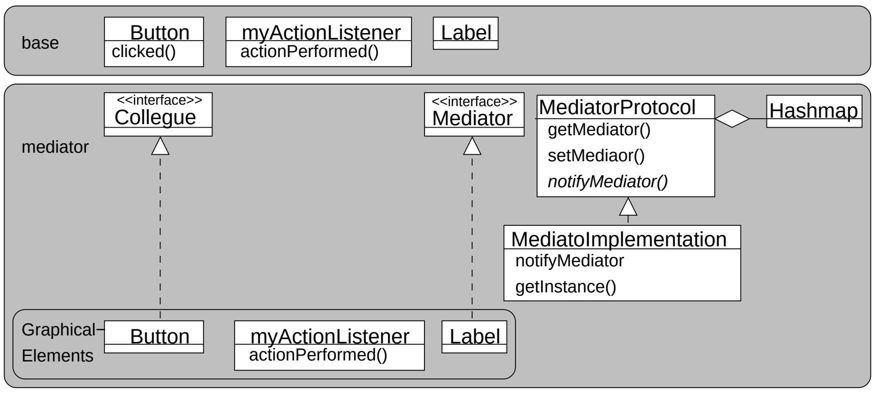

Figure 67. FOP implementation of the Mediator pattern.

Criteria	OOP	AOP	FOP
Cohesion	-	0	+
Variability	0	+	+

Table 14. Evaluation of the pattern Mediator

4.15 The Memento Design Pattern

4.15.1 Intention

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later [13].

4.15.2 Implementation

OOP solution. The design pattern Memento is applied to store the field-values of a `Counter` object, see Figure 68. The `Counter` object creates an object of type `CounterMemento` that contains the current values of the fields of the respective `Counter` object (in the method `createMementoFor`). The safed field-values of the `Counter` object can be restored by transferring the values back from the `CounterMemento` object to the `Counter` object (method `setMemento`). If the field-values of the `Counter` object have been changed in between, e.g., by the method `increment`, the fields are overriden by the values stored in the `CounterMemento` object by transfering the member values back to the `Counter` object.

Advantages The `Counter` object manipulates its corresponding `CounterMemento` object on its own and thus the manipulating `main` method is not aware of, i.e., are loosly coupled to different memento types and the internal representation of the object to store.

Disadvantages The interface of the `CounterMemento` object that is present to the `Counter` class is also available for any other object. Hence, the state can be manipulated from other classes and thus the encapsulation of the internal state (field-values) is broken (using C++ the `friend` statement can solve that problem).

AOP solution. The aspect `CounterMemento` is used to extract (method `createMementoFor`) the counters state and to restore (method `setMemento`) the counters state.

Advantages The field-values of an object can be saved by loosely coupled objects that do not know internals of the class to store.

Disadvantages The aspect `CounterMemento` is closely coupled to the `Counter` class because it determines the type of the object to store due to type casts. Thus the aspect is not reusable with other classes.

FOP solution. The creation and resetting of `Memento` objects is done by the singleton class `CounterMemento`, i.e., the methods `createMementoFor` and `setMemento` (Fig. 71). To enhance ex-

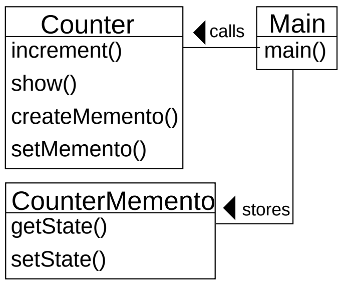

Figure 68. OOP implementation of the Memento pattern.

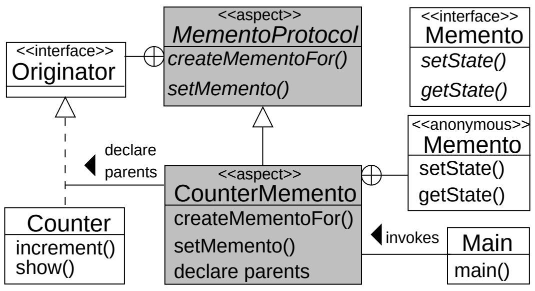

Figure 69. AOP implementation of the Memento pattern.

```
public void setMemento(Originator o, Memento m) {
 if (o instanceof Counter) {
  Integer integer = (Integer) m.getState();
  ((Counter)o).currentValue = integer.intValue();
 } else {
  throw new MementoException("Invalid originator");
 }
}
```

Figure 70. Resetting the field-values of a class to saved values in the aspect `CounterMemento`.

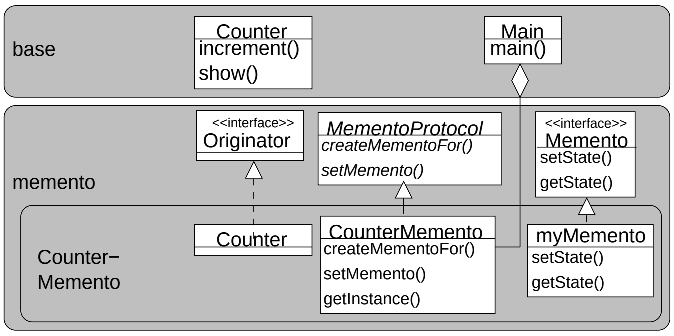

Figure 71. FOP implementation of the Memento pattern.

Criteria	OOP	AOP	FOP
Cohesion	-	0	+
Variability	0	+	+

Table 15. Evaluation of the pattern Memento

tensibility we extract the anonymous class `Memento` into the top-level class `MyMemento`.

Advantages The advantages of the AOP implementation hold for the FOP implementation.

Disadvantages The disadvantges of the AOP implementation hold for the FOP implementation.

4.15.3 Discussion

Cohesion. In the OOP implementation the variant methods storing the field-values of `Counter` objects into the memento objects are tangled within the `Counter` class.
In the AOP implementation the variant methods of storing the field-values of objects (e.g., `createMementoFor`) are separated from the code that is essential, e.g., `Counter.increment`, but methods that store and reset objects, e.g., of type `Counter`, are scattered across the aspects `MementoProtocol`, `CounterMemento` and the interface `Memento`.
In the FOP implementation the variant methods that store and reset field-values of objects are separated from the essential code of stored objects (e.g., `Counter.increment`) and the variant code is merged into one feature module MEMENTO.

Variability. In the OOP implementation the `Counter` class is tangled to the `CounterMemento` class due to `createMemento` method thus the `Counter` class depends on the class `CounterMemento`, i.e., the modules can not be composed in a flexible way.
In the AOP and FOP implementations the `Counter` class does not depend on the corresponding memento class (`Memento`) because the memento object that stores the field-values of a `Counter` object is created by the aspect `CounterMemento` (AOP) and the singleton class `CounterMemento` (FOP) respectively.

4.15.4 Summary

The aspect `MementoProtocol` only defines an empty interface (`Originator`). The only mechanism that is not OOP is one `declare parents` statement in the `CounterMemento` aspect and aspect methods to be defined by the subaspects.

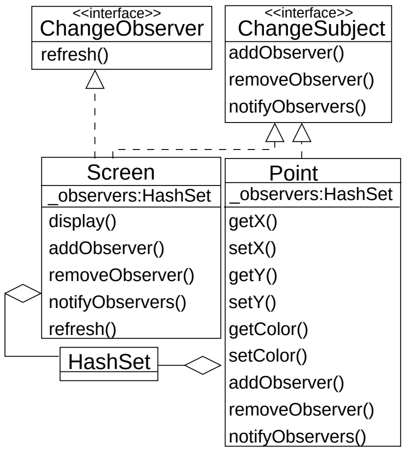

Figure 72. OOP implementation of the Observer pattern.

4.16 The Observer Design Pattern

4.16.1 Intention

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [13].

4.16.2 Implementation

OOP solution. The pattern is used to update `Screen` objects (the observers) after the position or the color of a `Point` object (the subject) has been changed (e.g., methods `setX`, `setColor`; Fig. 72). The method that changes the state of the subject object (of type `Point`), e.g., `setColor`, additionally invokes the method `notifyObservers`. The method `Screen.refresh` extracts the observers of a subject object from a hashset and invokes the method `refresh` of the oberving objects. This `refresh` method, e.g., `Screen.refresh`, updates the observer object (e.g., of type `Screen`).
The observed objects, e.g., of type `Point`, of an `ChangeObserver` object can be exchanged with respect to the `ChangeSubject` interface. The observing objects, e.g., of type `Screen`, of an observed object can be exchanged with respect to the `ChangeObserver` interface.

Advantages Different `ChangeObserver` objects, e.g., of type `Screen`, can be updated without changing the subject class, e.g., `Point` and thus the class is decoupled from the type and the number of observers to update.

Disadvantages If every observer object (e.g., of type `Screen`) is invoked every time the subject object changes (e.g., of type `Point`) the performance decreases due to the notification overhead.
Observers, e.g., `Screen` objects, can not be updated specific to their types, since that concrete type is not known to the subject class.
If the set of events of a subject object that are triggered to the observer objects should change code replication or invasive changes are necessary.
If the subject objects shall not update observer objects either code replication or invasive changes are necessary.

AOP solution. In the AOP implementation code of notifying observer objects is merged into the aspect `ObserverProtocol` (Fig. 73). The inheriting aspects `ColorObserver`, `Coordinate-Observer`, and `ScreenObserver` define the join points when to

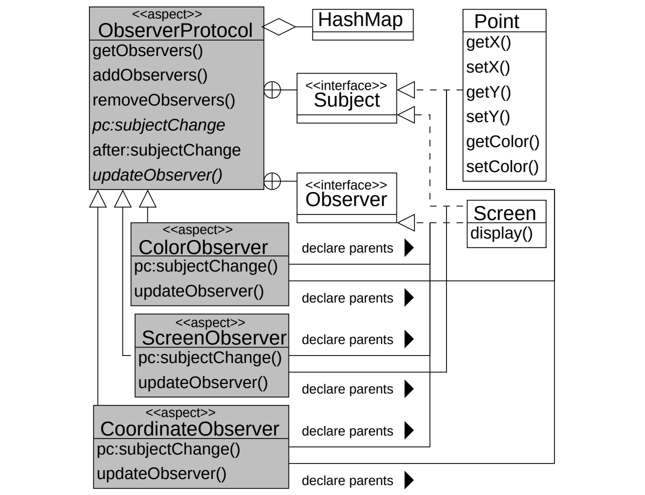

Figure 73. AOP implementation of the Observer pattern.

update observer objects by PCA.
The woven advice invokes the aspect method that updates the observers (`updateObserver`). This method uses a hashmap to retrieve the observers per subject and invoke their `update` method. The aspect method `addObserver` is used to assign subject objects to observer objects.

Advantages Changes to the code that updates observer objects (e.g., `notifyObservers`) do not demand for code replication of the observed classes, e.g., `Point`.

Disadvantages Introducing another observer type either causes code replication or invasive changes because the `updateObserver` methods rely that the observer type is `Screen`.

FOP solution. We present 2 FOP approaches for that pattern: soluation A (Fig. 74) is close to the AOP implementation. The methods that shall update observer objects, e.g., `Point.setColor`, are extended subsequently in the feature module OBSERVER to invoke the `updateObserver` method of the singleton classes, e.g., `ColorObserver`. These methods of the singleton classes perform the updates on the observer objects that are retrieved using a hashmap field `observers` of the singleton classes. Each singleton class updates observer objects specific to an observed issue (e.g., the class `ColorObserver` updates observers when the color of the subject changes).
Solution B (Fig. 74) can be applied if all observers of a `Point` object shall be notified after a change to the subject object (of type `Point`) appeared although the specific observer may not be interested in one specific notification issue. Therefore, the notification has to be evaluated inside the `notify` method of each observer, e.g., `Screen.notify`.

Advantages The Advantages of the AOP implementation hold for the FOP implementation.
Solution B improves performance because no hashmap has to be evaluated to get the list observers for an observed `Point` object.

Disadvantages Solution A may cause code replication for the method extensions of the observed class that invoke the `updateObserver` method. The extension by observer types demand for updating the `updateObserver` methods, since they rely on the type `Screen` to be the observer type due to type casts.

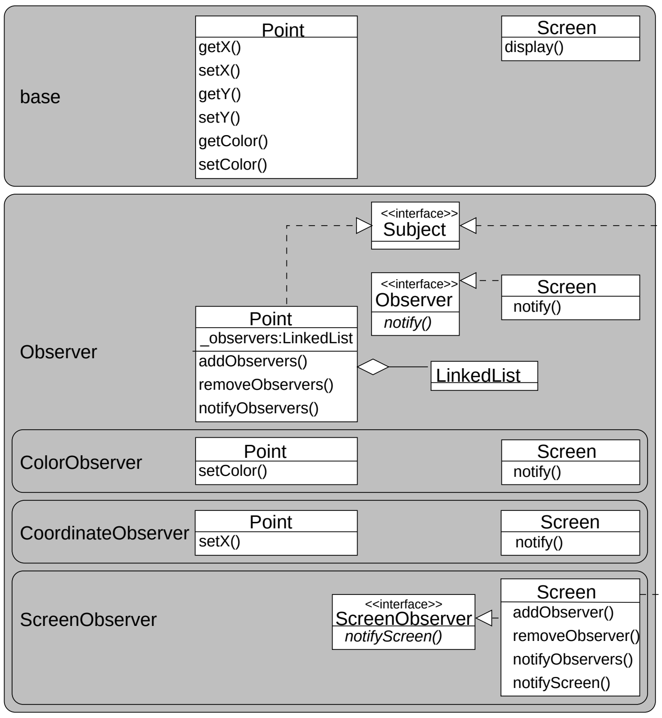

Figure 75. Alternative FOP implementation of the Observer pattern.

Criteria	OOP	AOP	FOP
Cohesion	-	0	+
Variability	0	+	+

Table 16. Evaluation of the pattern Observer

4.16.3 Discussion

Cohesion. In the OOP implementation the code that updates observer objects is scattered across the classes `Point`, `Screen`, `ChangeObserver`, and `ChangeSubject` and tangled with the code of the original concern of the respective classes.
In the AOP implementation the variant code of updating observer objects is detached into the aspects but scattered across the aspects `ObserverProtocol`, `ColorObserver`, `CoordinateObserver`, and `ScreenObserver`.
In the FOP implementation the variant code of updating observer objects, e.g., of type `Screen`, is detached and merged into the feature module OBSERVER.

Variability. In the OOP implementation the classes depend on each other, i.e., the classes `Screen` and `Point` depend on the `ChangeObserver` and `ChangeSubject` interfaces, thus building a monolithic system.
In the AOP and FOP implementations the update of observing objects can be omitted. Hence, the `Point` and `Screen` class do not depend on interfaces, like `Subject` or `Observer`, but can be omitted and exchanged.

4.16.4 Summary

A summary is given in Table 16.

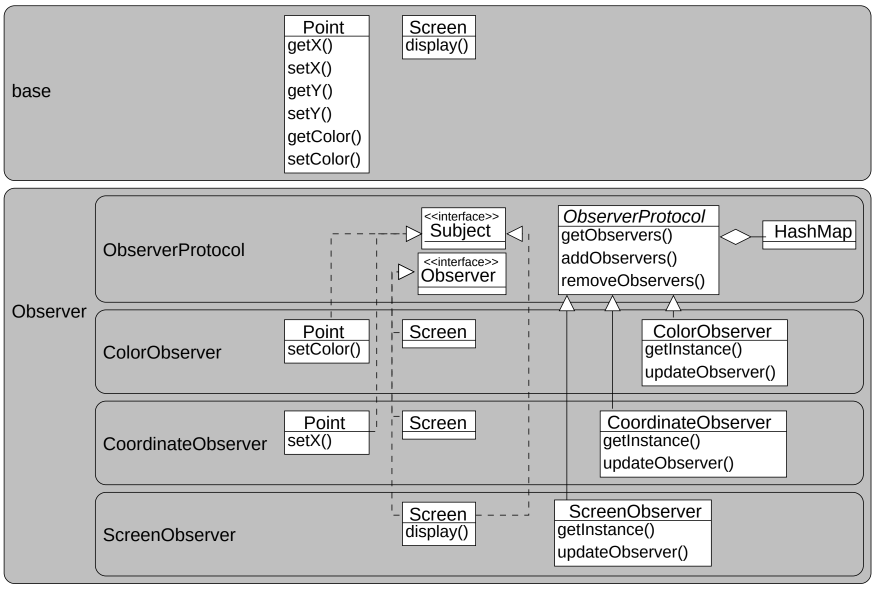

Figure 74. FOP implementation of the Observer pattern.

4.17 The Prototype Design Pattern

4.17.1 Intention

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype [13].

4.17.2 Implementation

OOP solution. The prototype design pattern has been applied to replicate objects of abstract implementation classes, i.e., `StringPrototypeA` and `StringPrototypeB`, without knowing the internal representations of the respective classes. For that, the both classes have to implement the interface `Clonable` and thus provide a method to copy its field-values to another object of the same class.

Advantages The code performing the replication of objects can vary for each variant class. Different objects can be replicated using the common interface `Cloneable`, i.e., clients, that replicate different objects do not have to know about the internal representations of the replicated objects.

Disadvantages The cloning of objects can hamper performance and resource consumption compared to object creation.
The code that clones objects of different types is scattered across the classes to clone.
If the classes `StringPrototypeA` and `StringPrototypeB` should not be able to replicate itself, i.e., should not implement the interface `Clonable`, the classes have to be changed invasively or replicated.

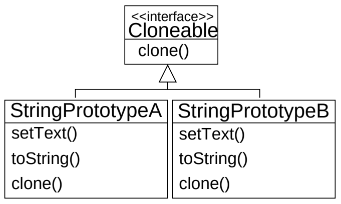

Figure 76. OOP implementation of the Prototype pattern.

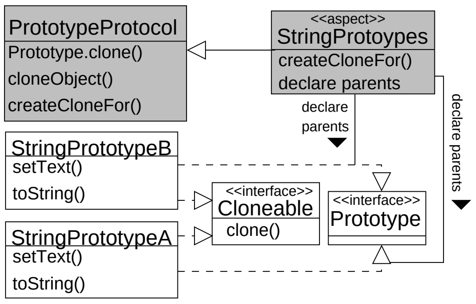

Figure 77. AOP implementation of the Prototype pattern.

AOP solution. The AOP implementation merges the code for cloning different classes into the aspects `PrototypeProtocol` and `StringPrototypes`. That method `PrototypeProtocol.cloneObject` replicates its parameter object or invokes the method `createCloneFor` of the aspect `StringPrototypes`, which replicates the object.

Advantages The advantages of the OOP implementation hold for the AOP implementation.
The code of replicating objects can be manipulated subsequently by replacing the aspects – that does not replicate the classes (`StringPrototypeA` and `StringPrototypeB`) of the objects to copy. Clients, that replicate different objects do not have to know about the internal representations of the objects to replicate.

Disadvantages The code replicating the class `StringPrototypeA` is tangled with the code of replicating the class `StringPrototypeB` thus decreasing variability.

FOP solution. The FOP implementation is close to the AOP implementation, see Figure 78. We merged the code replicating different objects into the method `PrototypeProtocol.cloneObject`.

Criteria	OOP	AOP	FOP
Cohesion	0	0	+
Variability	+	0	0

Table 17. Evaluation of the pattern Prototype

The methods `createCloneFor` and `cloneObject` replicate their parameter objects.

Advantages The advantages of the AOP implementation hold for the FOP implementation.

Disadvantages The code cloning different types of objects is tangled within the method `PrototypeProtocol.cloneObject` thus the code performing the replication of objects of different types can not vary for each type without code replication.

4.17.3 Discussion

Cohesion. We have to consider 2 issues:

- In the OOP implementation the code that replicates objects is scattered across the classes `Cloneable`, `StringPrototypeA`, and `StringPrototypeB`, and closely coupled to the main concerns of these classes.
  In the AOP implementation the code of replicating objects of different types is is scattered across the aspects and the variant aspects are not separated from the classes that are essential, e.g., `StringPrototypeA`. In the FOP implementation the single feature module PROTOTYPING merges the variant mixin classes that implement the replication of objects.
- In the OOP implementation the code of the classes `StringPrototypeA` and `StringPrototypeB` is merged in the respective classes. In the AOP implementation the code of these classes is scattered across the aspects and coupled within. In the FOP implementation the code of the classes `StringPrototypeA` and `StringPrototypeB` is scattered across the feature modules BASE and PROTOTYPING.

Variability. In the OOP implementation the code replicating an object of a specific class (the `clone` method) can be exchanged with respect to the code replicating objects of other classes. In the AOP and FOP implementations the code replicating one class is tangled to the code replicating another class inside the method `PrototypProtocol.cloneObject`. Hence, if the code replicating objects of one class should be exchanged, invasive changes or code replication are necessary.

4.17.4 Summary

A summary of the evaluation is depicted in Figure 17.

4.18 The Proxy Design Pattern

4.18.1 Intention

Provide a surrogate or placeholder for another object to control access to it [13].

4.18.2 Implementation

OOP solution. The clients of proxy classes each refer to an object of type `OutputImplementation` (Fig. 79) to invoke different methods. This referenced object and its methods respectively (e.g., `safeRequest`) are shielded by the Proxy objects (of the type `RequestBlocker` and `RequestCounter`) from access. If the Proxy classes are instantiated instead of the class `OutputImplementation`, the `RequestBlocker` and `RequestCounter` objects intercept and analyze requests to the `OutputImplementation` objects. Since the Proxy objects implement the same interface as the

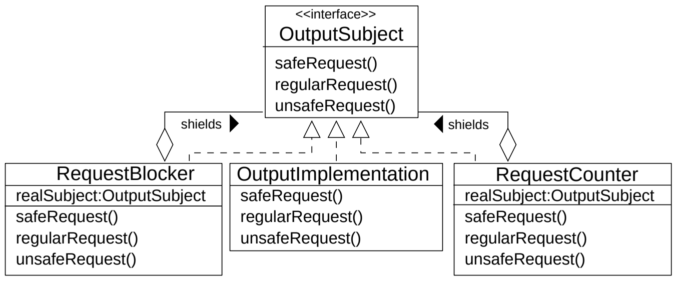

Figure 79. OOP implementation of the Proxy design pattern.

```
1 protected pointcut requests():call(*
   OutputImplementation.safeRequest(..));
2 private pointcut requestsByCaller(Object
   caller):requests() && this(caller);
3 Object around(Object caller, Subject subject):
   requestsByCaller(caller) && target(subject) {
4  if (! isProxyProtected(caller, subject,
     thisJoinPoint) )
5    return proceed(caller, subject);
6  return handleProxyProtection(caller, subject,
     thisJoinPoint);
7 }
```

Figure 80. Caller analysis in AOP advice.

class `OutputImplementation`, clients (e.g., the `main` class) are not affected if the `OutputImplementation` object is replaced by a Proxy object of type `OutputSubject`. The Proxy objects refer to the respective shielded object by an object reference that is instantiated as needed.

Advantages Costs for instantiating and manipulating the output implementation object can be delayed until its properties are accessed. Proxy objects of the classes `RequestBlocker` or `RequestCounter` can be applied, omitted, and exchanged at runtime. Proxies can hide complexity of accessing an object, e.g., a remote object.

Disadvantages For all methods of a shielded object indirections are introduced by the shielding Proxy objects and thus the performance decreases, e.g., indirections are introduced by proxy objects of type `RequestBlocker` and `RequestCounter`.
Hiding methods, e.g., `OutputImplementation.safeRequest`, subsequently by introducing a new Proxy class affects the classes that invoke methods of the `OutputImplementation` object (e.g., the class `main`) because the calling class have to instantiate this new Proxy class instead of the class `OutputImplementation`.
If the caller of the output implementation object has to be analyzed by the Proxy objects the calling object has to provide itself as a parameter. This parameter is not necessary if no Proxy object analyzes the caller objects. Thus the caller classes, e.g., the class `main`, are affected, if a Proxy analyzes the caller – that causes close coupling.

AOP solution. Proxy shielding and redirection of method invocations, e.g., for the method `OutputImplementation.safeRequest`, is applied by intercepting respective method *calls* using PCA.
The caller object of a method of the `OutputImplementation` object is gathered through the pointcut expressions (Fig. 80, Lines 1–2) and is analyzed for its type by the advice of the pointcut `requestsByCaller` (Line 3–7) by invoking the method `isProxyProtected` (Line 4). If the analysis in this aspect method

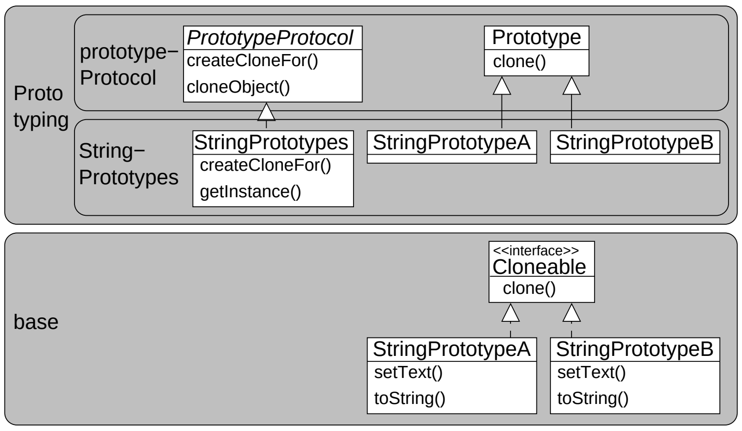

Figure 78. FOP implementation of the Prototype pattern.

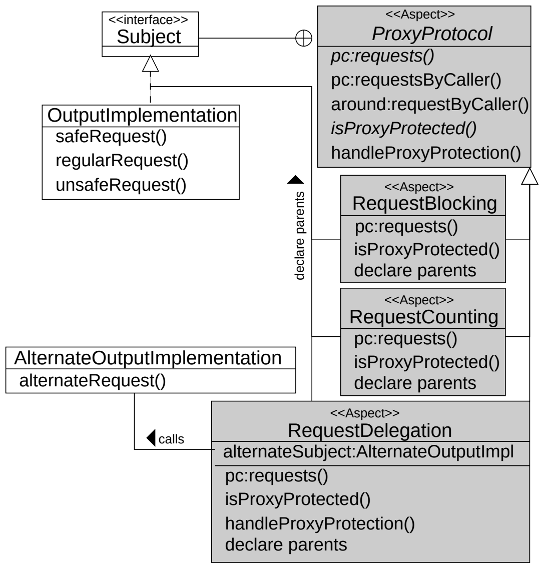

Figure 81. AOP implementation of the Proxy design pattern.

succeeds the output implementation object is invoked by the aspect (Line 5). If the method `isProxyProtected` fails the call is denied by invoking the empty method `handleProxyProtection` (Line 6).

Figure 81 depicts the implementation proposed by Hannemann et al. The output implementation class `OutputImplementation` is manipulated by the class `main`. The aspect `ProxyProtocol` includes the advice performing the analysis-dependent forwarding to the shielded methods (Fig. 80, Lines 3–7). The analysis of the caller object and the definition of methods to shield (pointcut `requests`) is implemented in the subaspects, e.g., `RequestBlocking`.
While the aspects `RequestBlocking` and `RequestCounting` are simply forwarding or denying (by returning a null value) calls to the shielded methods of the `OutputImplementation` object, the aspect `RequestDelegation` forwards these calls to a different class (`AlternateOutputImplementation`) instead.

Advantages The advantages of the OOP implementation, e.g., delaying the time of instantiation of the `OutputImplementation` object, hold for the AOP implementation.
The application of caller analysis and method shielding can be applied without changes to the `main` class and without changes to the `OutputImplementation` class.
If the call to one method of the `OutputImplementation` object (e.g., `safeRequest`) is analyzed, the analysis code is introduced around this single method. That is, other methods of the object (e.g., `regularRequest`) stay unaffected and can be accessed directly by the `main` class. This prevention of indirections increases performance by omitting empty forwarding methods.

Disadvantages There are no obvious disadvantages associated to this pattern implementation.

FOP solution. FOP does not allow to extend method calls in general and thus FOP implementations can not analyze the object straightforwardly that calls methods of the shielded `OutputImplementation` objects, e.g., the `main` class. To cope with this problem, we introduced one mixin and one method extension per shielded method (Fig. 82), e.g., for the method `OutputImplementation.safeRequest` we introduced one mixin and one method extension. The mixin associated to a shielded method, e.g., `safeRequest`, extends the signature of this method in the mixin class `proxyProtocol.OutputImplementation` to accept the calling object as an additional parameter – this extended method is *caller-aware* and only forwards calls to the refined caller-*un*aware methods of the feature module BASE. Subsequently, we refine the caller-aware methods to invoke the `isProxyProtected` methods and thus to analyze the additional parameter (the caller), e.g., in the feature module REQUESTBLOCKING. Hence, these *caller-aware* methods, that include the additional parameter, are shielded instead of the originals.

Advantages The code to analyze callers of methods is merged into the feature module PROXYPROTOCOL.

Disadvantages In Java access to caller unaware methods cannot be restricted by additive changes.[12] C++ allows to decrease the accessibility of the caller-unaware methods, e.g., `base.OutputImplementation.safeRequest`.
The class (`main`) that calls methods of the `OutputRepresentation` object has to attend the caller analysis by invoking the methods with the additional parameter, i.e., the caller-aware methods.
Due to superimposition the namespace of different feature modules are merged and methods that are specific for one module and shall not be overridden have to be renamed. For in-

[12] Overriding methods in subclasses can not decrease the visibility declared for them by their respective overridden methods of the superclass.

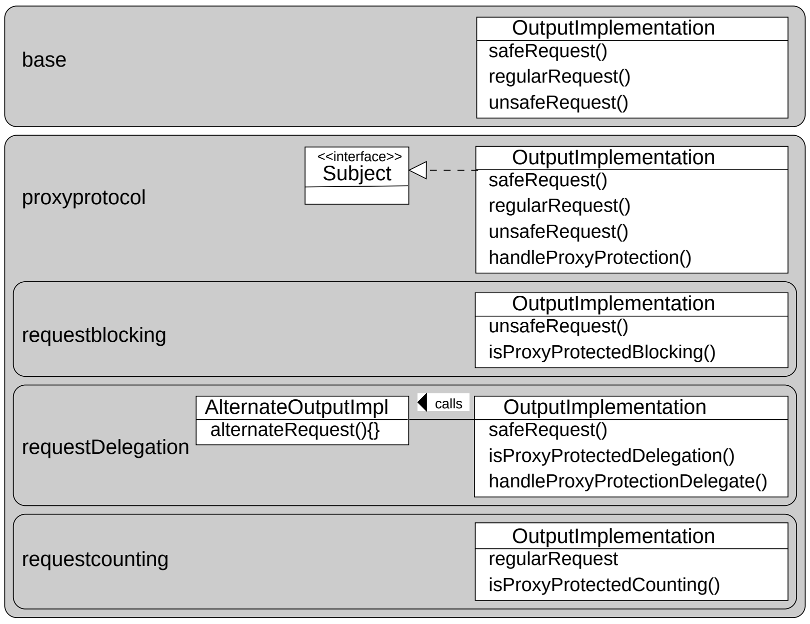

Figure 82. FOP design of the Proxy design pattern.

stance, we renamed the method `OutputImplementation.is-ProxyProtected` of the feature module REQUESTBLOCKING into `OutputImplementation.isProxyProtectedBlocking` to prevent overriding, e.g., by the mixin `requestCounting.-OutputImplementation.isProxyProtected`.

Changing the number of parameters of methods causes a problem in extending these methods, equivalently to the constructor problem, that has been solved for C++ in [11].

4.18.3 Discussion

Cohesion. In the OOP implementation the analysis of method callers and the forwarding implementation are detached from the `OutputImplementation` class but scattered across all proxy classes, e.g., `RequestCounter`. In the AOP implementation the caller analysis code is scattered for different methods of the `OutputImplementation` class across all aspects. In the FOP implementation the code of the caller analysis of the output implementation objects is merged in one feature module PROXYPROTOCOL.

Variability. To evaluate the variability of the implementations, we have to consider two aspects:

- In the OOP implementation the class `OutputImplementation` depends on the Proxy interface (`OutputSubject`).
  In the AOP implementation the class `OutputImplementation` does not depend on the interface `OutputSubject` because the method shielding is introduced additively and directly into the class.
  FOP also removes dependencies between the the class `Output-Implementation` and the interface `OutputSubject`.

- In the OOP implementation flexible caller analysis demands for either code replication or invasive changes of the `main` class because it has to instantiate the proxy class, e.g., `RequestBlocker`, instead of the class `OutputImplementation`. The AOP implementation allows to reuse the `main` class because the caller analysis is applied without affecting this class invasively. FOP needs to adapt the objects that invoke shielded methods, e.g., `OutputImplementation.safeRequest` to provide an additional parameter. Hence, the `main` class is not reusable.

Criteria	OOP	AOP	FOP
Cohesion	0	0	+
Variability	-	+	0

Table 18. Summarized evaluation of the Proxy design pattern.

4.18.4 Summary

The OOP implementation introduces different indirections for all methods of the class when different proxy objects shield an object. The AOP implementation avoids indirections for methods that are not shielded because PCA extends single methods of a class. The FOP implementation introduces one indirection for all potentially shielded methods of a class.

In the OOP and AOP implementations the analysis of callers may vary at runtime, e.g., depending on the object (OOP) or depending on the control flow (AOP). In FOP caller analysis is applied for all instances and calls to the shielded methods.

4.19 The Singleton Design Pattern

4.19.1 Intention

Ensure a class only has one instance, and provide a global point of access to it [13].

4.19.2 Implementation

OOP solution. Hannemann et al. applied the Singleton pattern to limit the number of objects of a printer class (`PrinterSingleton`) instantiated at runtime. Therefore, the `PrinterSingleton` class instantiates a static reference of the own type and refers to this element subsequently every time the class should be instantiated – the method `instance` is used to retrieve that single instance.
The subclass `PrinterSubclass` overrides the method `instance` and omits the limitation regarding the count of instances, i.e., the methods returns a *new* object every time it is called.

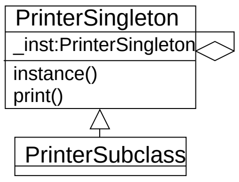

Figure 83. OOP implementation of the Singleton pattern.

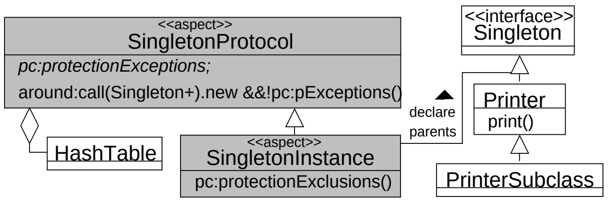

Figure 84. AOP implementation of the Singleton pattern.

Advantages Multiple instances of the class `PrinterSingleton` can be prevented or the count of instances can be limited.
The class `PrinterSingleton` merges synchronization effort.

Disadvantages If the class `PrinterSingleton` should not act as a singleton, the clients (the `main` method) have to call the constructor of the class instead of calling the `instance` method thus clients have to be adapted or the `instance` method has to be manipulated.

AOP solution. In the AOP implementation every call to the constructor of the singleton class (`Printer`) is intercepted by PCA using around advice. If an object of the according class (`Printer`) exists in a hashmap field of the aspect `SingletonProtocol`, this object is returned by advice instead of a newly created object – the constructor is not called. If there is no object of the class to instantiate in the hashtable, an object of the respective class is created and is stored in the hashtable to be used for subsequent requests.

Advantages The advantages of the OOP implementation are kept. The `main` method is not affected whether the class `Printer` acts as a singleton or not.

Disadvantages The manipulation of one single instance by multiple clients may demand for additional synchronization code.
The performance decreases due to the evaluation of the hashtable in the advice to find existing objects of a class.
Due to the empty interface `Singleton` several type casts are necessary that are error prone.

FOP solution. Our FOP implementation is close to the OOP implementation (Fig. 85). The classes `Printer` and `PrinterSubclass` implement a method `getInstance` that controls the creation of instances for these classes.
The method `getInstance` of the mixin class `SingletonInstance.Printer` proves, whether an object of the class `Printer` exists (and is stored in the static reference of the `Printer` class). If so, the existing object is returned. Otherwise, an object is created and stored in the static reference. Then this instantiated object is returned.
The `PrinterSubclass` is refined to create new `PrinterSubclass` objects every time the method `getInstance` is called.

Advantages The advantages of the AOP implementation hold for the FOP implementation but the synchronization effort is merged into the feature module SINGLETONINSTANCE instead of aspects.

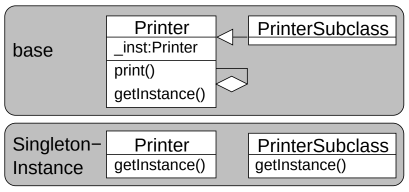

Figure 85. FOP implementation of the Singleton pattern.

Criteria	OOP	AOP	FOP
Cohesion	-	0	+
Variability	0	+	+

Table 19. Evaluation of the pattern Singleton

Disadvantages Since decrease of accessibility of methods and constructors is not possible in Java subsequently, the limited view of instances has to be prepared to limit the count of objects for one class. Hence, we implemented the object creating method `getInstance` from beginning. C++ allows to reduce the visibility of class members by subclasses, thus the preparation of the pattern is not necessary in C++ based approaches of FOP but changes changes affect the clients.
The application of the pattern to different classes causes code replication of the `getInstance` method and the static reference.

4.19.3 Discussion

Cohesion. In the OOP implementation the code limiting the number of instances is scattered across the classes `PrinterSingleton` and `PrinterSubclass`. Furthermore, the essential members of the class `PrinterSingleton`, i.e., the method `print`, are closely coupled with the variant members, i.e., the method `instance`.
The AOP implementation detaches the code to limit the number of instances from the class but scatters the code across the aspects `SingletonProtocol` and `SingletonInstance`.
In the FOP implementation the variant code (the method `instance`) is detached from the essential elements of the class and is merged into the one feature module SINGLETONINSTANCE.

Variability. In the OOP approach the class `PrinterSingleton` only can be used as singleton. In the AOP and FOP implementations the `Printer` classes can be used as singleton classes and as normal ”multi-object” classes.

4.19.4 Summary

The question concerning this adaptibily is, whether it is meaningful to apply the limitation regarding the count of instances for one class obliviously.

FOP does not allow to intercept calls to constructors inside complex methods since the extension of a constructor using refinements does not prevent the creation of an object.

4.20 The State Design Pattern

4.20.1 Intention

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class [13].

4.20.2 Implementation

OOP solution. The pattern is applied to adapt the implementation of a queue object (of class `Queue`; Fig. 86) based on the number of elements in the queue, e.g., the insertion of additional

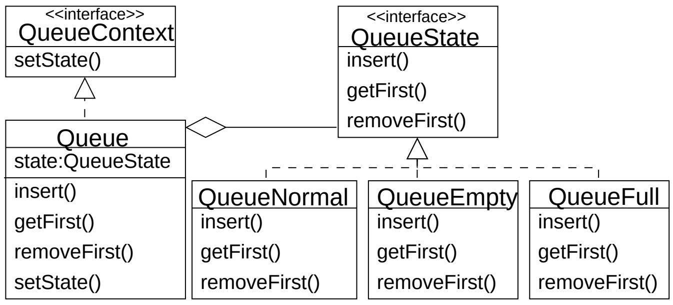

Figure 86. OOP implementation of the State pattern.

elements fails if the queue is full. The variant method implementations of the queue, e.g., of method `insert`, are detached into referred classes, e.g., `QueueFull`. The different classes implementing methods with respect to the state of the `Queue` (`QueueEmpty`, `QueueNormal`, `QueueFull`) can be exchanged with respect to the interface `QueueState`.
The `Queue` object forwards requests (`insert`, `getFirst`, or `removeFirst`) to the object representing the current state and is referred to by the `Queue` object. The state object implements the methods according to the type of the state class, e.g., state objects of type `QueueFull` reject element insertion. By exchanging the state object, the `Queue` object behaves differently for clients.

Additionally, the state objects, e.g., of type `StateFull`, change the state object that the `Queue` object refers to, thus replacing themselve, e.g., by a `QueueEmpty` object.

Advantages All behavior associated with an internal state of the queue, e.g., the queue is full, is merged inside the respective state class, e.g., `QueueFull`.
The transition between states (e.g., state `QueueFull` becomes `QueueNormal` after object deletion) is depicted explicitely by the changes of the state objects.
The state classes, e.g., `QueueFull` can be reused to represent the state of other classes.

Disadvantages The code of changing the state of a `Queue` object is scattered across the state classes `QueueNormal`, `QueueEmpty`, and `QueueFull`.
Each state class is closely coupled to the state class that succeeds after an operation.
Changing the schedule of state classes that succeeds each other demands either for invasive changes or code replication.

AOP solution. In the AOP implementation the code exchanging the state objects of a `Queue` object is merged in the aspect `QueueStateAspect`. The aspect advises different join points to assign a new state object of a specific type to the `Queue` class.
An example is given Figure 87. After the method `insert` has been invoked, state of the `Queue` object is updated (Lines 3–4 and 7–9) based on the current state. If the queue was empty before insertion (Line 2) the insertion of an element causes the state to be normal; if the queue was filled normally (Line 5)and after the insertion the `Queue` can not store additional elements (Line 6), the state of the `Queue` object is set to be full (Line 9).

Advantages The transition of state objects is separated from the state classes and thus the transition definitions can be exchanged without code replication or invasive changes. Hence, the integration of a new state class is possible without manipulating existing state classes.

```
1   after(Queue queue, QueueState qs, Object arg):
      call(boolean QueueState+.insert(Object)) &&
      target(qs) && args(arg) && this(queue) {
2     if (qs == empty) {
3       normal.insert(arg);
4       queue.setState(normal);
5     } else if (qs == normal) {
6       if (normal.first == normal.last) {
7         full.items = normal.items;
8         full.first = normal.first;
9         queue.setState(full);
10      }
11    }
12  }
```

Figure 87. Changing the state of a `Queue` object by advice.

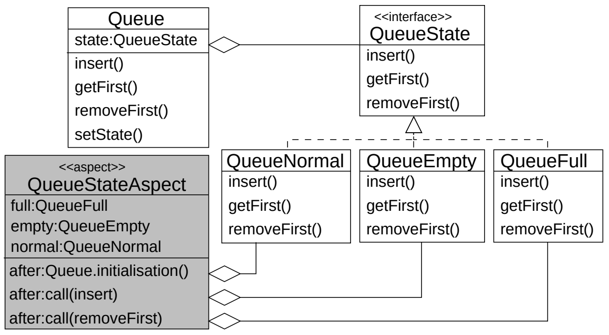

Figure 88. AOP implementation of the State pattern.

Disadvantages Changes to the ordering of state objects causes the replication of the aspect `QueueStateAspect`. The aspect `QueueStateAspect` is coupled closely to the `Queue` class, thus lacks reuse if the state classes are used in different situations.
Modularizing the state transitions lacks the aim of the pattern to separate behavior of different states [13] because the code specific to different states is tangled within the aspect `QueueStateAspect`.

FOP solution. In the FOP implementation the transitions between states of an `Queue` object are implemented as extensions of the `Queue` methods that may cause a change of state, i.e., methods, like `QueueState.Queue.insert`, adapt the state of the current `Queue` object by exchanging its assigned state object. In contrast to the OOP and AOP implementation the `Queue` class is extended to adapt itself but is not adapted by the `Queuestate` classes (e.g., `QueueNormal`) or the aspect (`QueueStateAspect`).

Advantages The advantages of the AOP implementation hold for the FOP implementation.

Disadvantages The refinement `QueueState.Queue` is closely coupled to the `Queue` class thus lacks reuse if the state classes and their transitions are applied in other situations.
Modularizing the state transitions lacks the aim of the pattern to separate behavior of different states [13]. The code specific to different states is tangled within the feature module QUEUESTATE.

4.20.3 Discussion

Cohesion. We have to consider two aspects to evaluate cohesion of these pattern implementations:

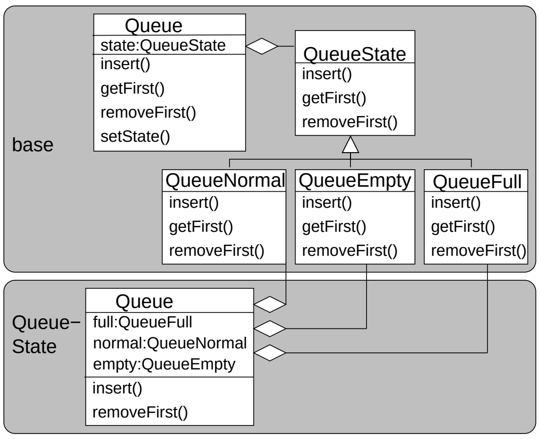

Figure 89. FOP implementation of the State pattern.

- In the OOP implementation the transition code is scattered across all state classes. In the AOP and FOP implementations the transition code is modularized in the aspect `QueueStateAspect` (AOP) and the feature module QUEUESTATE (FOP) respectively.
- In the OOP implementation the code of the variant state transition is closely coupled to the state classes. In the AOP implementation the variant code of state transition is detached from the state classes but the module that implements the variant transitions is not separated from the essential classes of `Queue` and `QueueState`. The FOP implementation separates the variant transition code from the essential state code into the mixin class `QueueState.Queue` and the variant mixin classes are separated from the essential classes, e.g., `Queue`.

Variability. We have to consider 2 aspects:

- In the OOP implementation every state class, e.g., `QueueFull`, depends on the succeeding state class, e.g., `QueueNormal`. Since this restriction holds for all state classes, no state class can be exchanged without code representation or invasive changes. In the AOP and FOP implementations the state classes, e.g., `QueueFull` do not depend on each other and thus can be exchanged.
- In the OOP implementation the definition of the transition for every state class (e.g., state `QueueFull` follows `QueueNormal` after element insertion) can be exchanged without corrupting the code associated to other classes. In the AOP and FOP implementations the transitions between all states are tangled within the aspect `QueueStateAspect` (AOP) and the `QueueState.Queue` class (FOP) respectively and thus exchanging the definition regarding one state class causes code replication or invasive changes for the definitions regarding the other classes.

In summary the OOP, AOP, and FOP implementations are equivalent regarding the variability in this pattern.

4.20.4 Summary

Modularizing the state transitions as in AOP and FOP lacks the aim of the pattern to separate behavior of different states [13].

Criteria	OOP	AOP	FOP
Cohesion	-	0	+
Variability	0	0	0

Table 20. Evaluation of the pattern State

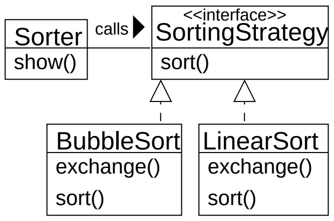

Figure 90. OOP implementation of the Strategy pattern.

4.21 The Strategy Design Pattern

4.21.1 Intention

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [13].

4.21.2 Implementation

OOP solution. Hannemann et al. proposed to assign different algorithms, e.g., bubble sort or linear sort, to a sorting component (`Sorter`, Fig. 90). Hence, the different algorithms are separated from the `Sorter` class into *strategy* classes each, e.g., `BubbleSort` and `LinearSort`. The strategy classes can vary with respect to the referred `SortingStrategy` interface at runtime. The `Sorter` class refers to a strategy object and forwards calls of the `sort` method to the strategy object, i.e., based on the dynamic type of the referred strategy object, e.g., `BubbleSort`, different implementations are executed for the forwarded method (`sort`) at runtime.

Advantages Variant algorithms, e.g., linear sort, i.e., method implementations, are modularized in classes, e.g., `LinearSort`, and can vary at runtime for a given `Sorter` object. Conditional statements to choose the proper implementation are prevented, e.g., for method `sort`, since the implementations for methods are exchanged by assigning another strategy object to the `Sorter` object.

Disadvantages The strategy objects, e.g., of type `BubbleSort`, only can manipulate members of the `Sorter` class that are declared as *public*.
The forwarding of method calls by the `Sorter` class decreases performance.
The classes that use the `Sorter` object to sort elements assign the appropriate strategy objects to the `Sorter` object. Hence, the client (`main`) is coupled to the different strategies that are possible for the `Sorter` class.
Classes that should be applied as strategies for the `Sorter` class have to implement the common `SortingStrategy` interface.

AOP solution. The classes that implement different strategies, e.g., `BubbleSort`, are assigned to the `Sorter` objects using a hashmap inside the aspect `StrategyProtocol`. A hook method `sort` is introduced into the `Sorter` class. Calls to this hook method are intercepted by PCA. This PCA redirects the calls of the method `Sorter.sort` to the sorting methods `BubbleSort.sort` or `LinearSort.sort`.

Advantages The advantages of the OOP implementation hold for the AOP implementation.

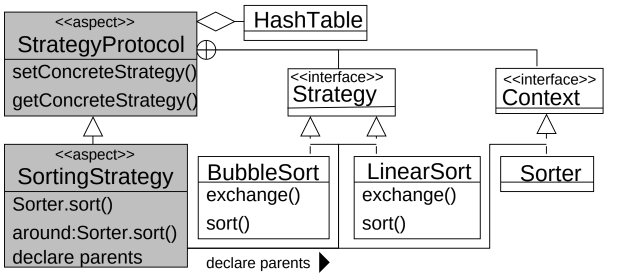

Figure 91. AOP implementation of the Strategy pattern.

```
1   int[] around(Sorter s, int[] numbers): call(int[]
      Sorter.sort(int[])) && target(s) && args(numbers)
      {
2     Strategy strategy = getConcreteStrategy(s);
3     if (strategy instanceof BubbleSort) {
4       ((BubbleSort)strategy).sort(numbers);
5     } else if (strategy instanceof LinearSort) {
6       ((LinearSort) strategy).sort(numbers);
7     } else {
8       // Invalid strategy: could throw an exception
          here
9     }
10    return numbers;
11  }
```

Figure 92. Changing the state of a `Queue` object by advice.

The `Sorter` class does not depend on the `Strategy` interface, because the variant code of the method `sort` is detached into the aspect `SortingStrategy`.
Any class that implements the method `sort` can be used to act as a sorting strategy for the `Sorter` class, i.e., the class does not have to be subtype of a common `Strategy` interface.

Disadvantages The indirection of the hashmap evaluation to get the associated strategy object decreases performance.
The clients of the `Sorter` object, e.g., the `main` method, have to assign the appropriate strategy, e.g., `BubbleSort`, to each `Sorter` object thus they are closely coupled to the `Strategy` objects.

Since the classes that implement the different strategies do not have to fulfill a common interface, the aspect `SortingStrategy` has to analyze the types of strategy objects at runtime thus decreasing performance, this is depicted in Figure 92 Lines 3 and 5 to perform the appropriate actions. (Conditional statements to select the strategy to invoke were aimed to be omitted by Gamma et al. [13] but occur in Lines 3 and 5 of the Figure 92.)

FOP solution. We present 2 FOP solutions for that pattern: Our first solution is close to the AOP implementation (Fig. 93). The strategy object, e.g., of type `BubbleSort`, is associated to the `Sorter` object by a key-value pair of the hashmap of the singleton class `StrategyProtocol`. The mixin `SortingStrategy.Sorter.sort` forwards calls to the method `sort` of the strategy object which performs the sort.
Solution B is depicted in Figure 94. This implementation is applicable, if the sorting strategy for all `Sorter` objects can be assigned at compile time, i.e., the sorting strategy shall not be exchanged at runtime. The sorting strategies, e.g., linear sort or bubble sort, are chosen by choosing the appropriate feature module, e.g., BUBBLESORT.

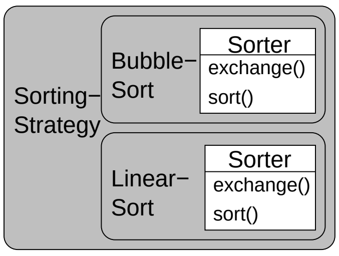

Figure 94. Alternative FOP implementation of the Strategy pattern.

Criteria	OOP	AOP	FOP
Cohesion	0	0	+
Variability	0	+	+

Table 21. Evaluation of the pattern Strategy

Advantages The advantages of the OOP and AOP implementations hold for the FOP implementation.
In solution B the clients of the sorter objects, e.g., the `main` method, are decoupled from the strategies of the `Sorter` objects since they do not have to assign the different sorting strategies to the `Sorter` objects – i.e., they do not have to know them.
In solution B the performance is improved because no hashmap has to be evaluated and the methods, e.g., `sort` and `exchange`, do not have to be bound dynamically. In solution B the strategy methods of the strategy objects, e.g., `sort`, can access all members of the `Sorter` class including members declared as `private` or `protected`.[13]

Disadvantages For solution A the disadvantages of the AOP implementation hold.

4.21.3 Discussion

Cohesion. In the OOP implementation different implementations for one method of the `Sorter` class are scattered across the classes `SortingStrategy`, `BubbleSort`, and `LinearSort`. In the AOP implementation the code regarding different `Sorter` algorithms is scattered across the classes `BubbleSort`, `LinearSort` and the aspects `StrategyProtocol` and `SortingStrategy`. In the FOP implementation the code regarding different strategies for the `Sorter` class is merged in the feature module DIFFERENTSTRATEGIES.

Variability. In the OOP implementation the classes to be used as strategies for the `Sorter` class are restricted to those that are subtype of the interface `SortingStrategy`. In the AOP and FOP implementations the objects of every class can be assigned to instantiate stragegy objects for `Sorter` objects.

4.21.4 Summary

A summary is given in Table 21.

4.22 The Template Method Design Pattern

4.22.1 Intention

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure [13].

[13] If mixin layers are used to implement FOP, the strategy methods, e.g., *sort*, can not access private members of the prior class refinements. Jampacks bypass this restriction.

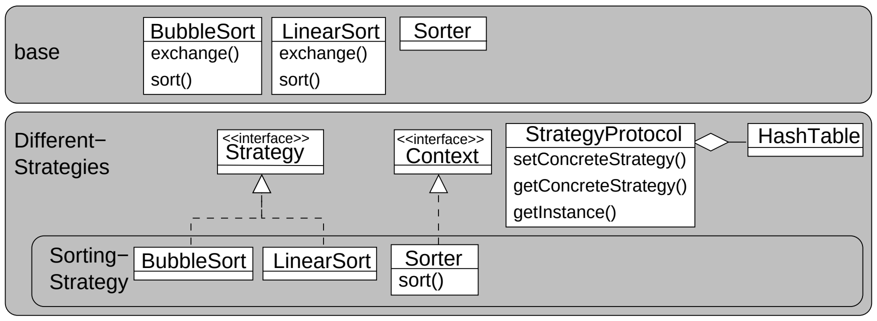

Figure 93. FOP implementation of the Strategy pattern.

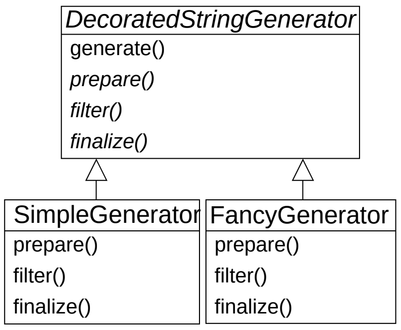

Figure 95. OOP implementation of the pattern Template Method.

4.22.2 Implementation

OOP solution. Hannemann et al. applied the pattern to compose the complex operation (`generate`) of `String` transformation out of atomic operations. The complex operation `generate` performs different atomic operations which can be exchanged with respect to the interface `DecoratedStringGenerator`. For example, the atomic operation `filter` may be implemented in a simple or fancy way. The different variants of the atomic operations, e.g., `SimpleGenerator.filter` and `FancyGenerator.filter`, are defined in sub classes `SimpleGenerator` and `FancyGenerator` of the class `DecoratedStringGenerator`. By selecting the instantiated class, i.e., `FancyGenerator` or `SimpleGenerator`, at instantiation time, the variants of the atomic operations (e.g., `filter`) defined in that class are used as steps for the compound operation `generate`.

Advantages Changing the atomic operations of an the complex `generate` operation does not effect the implementation of the `generate` method.

Disadvantages The variants associated to the complex operation `generate` have to be anticipated by invoking hook methods of atomic operations.

If the complex operation `generate` should be replaced the class `DecoratedStringGenerator` has to be extended or replaced thus causing code replication or invasive changes.

AOP solution. In the AOP implementation the variant complex operation `generate` is introduced by the aspect `Generating` using ITD, see Figure 96.

Advantages The advantages of the OOP implementation hold for the AOP implementation.
The composed operation can vary without replicating the primitive operations.

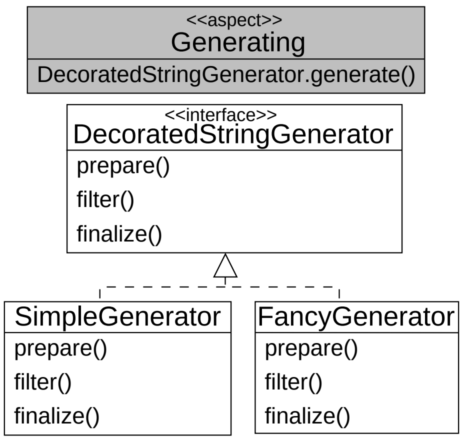

Figure 96. AOP implementation of the pattern Template Method.

Disadvantages Hook methods for the atomic operations are still necessary.

FOP solution. We present two solutions for the implementation of that pattern: In solution A the variant complex operation is separated from the implementations of the atomic operations in the feature module TEMPLATEMETHOD, see Figure 97.
Solution B (Fig. 98) is applicable if the atomic operation, e.g., `filter`, to compose the complex operation `generate` can be chosen at compile time.

Advantages The advantages of the OOP and AOP implementation hold for the FOP implementations A and B.
Solution B decreases the number of virtual methods in C++ since the different variants of methods are introduced directly into the class without inheritance. That improves the performance and the resource consumption of the software [10].
Solution B does not demand for hook methods.

Disadvantages For solution A hook methods are still needed to apply different primitive operations, e.g., `filter`, to the complex operation `generate`.

4.22.3 Discussion

Cohesion. All implementations of the Template Method design pattern are equivalent regarding cohesion. In all classes the complex operation is decoupled from the concrete implementation of the primitive operations. In all techniques the implementations of the different variants of the primitive operations are separated from each other.

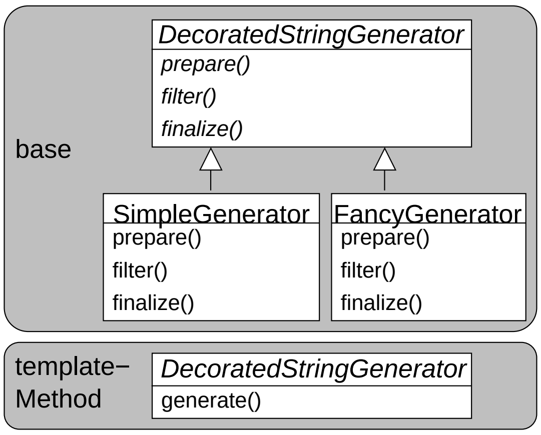

Figure 97. FOP implementation of the pattern Template Method.

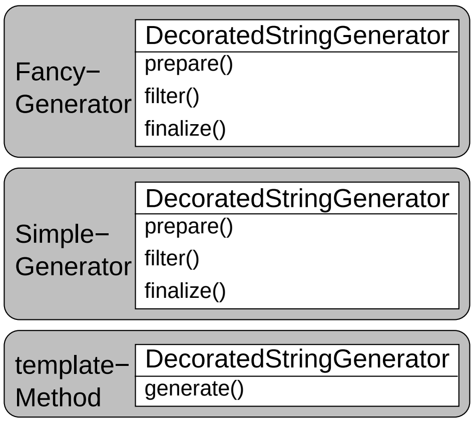

Figure 98. Alternative FOP implementation of the pattern Template Method.

Criteria	OOP	AOP	FOP
Cohesion	0	0	0
Variability	0	+	+

Table 22. Evaluation of the pattern Template Method

Variability. In the OOP implementation the primitive operations, e.g., `filter`, are tangled with the complex operation `generate` due to inheritance. The AOP and FOP implementations decouple the primitive operations from the complex method `generate` and thus the `generate` method can be exchanged.

4.22.4 Summary

A summary is given in Table 22.

4.23 The Visitor Design Pattern

4.23.1 Intention

> Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates [13].

```
1  public class BinaryTreeNode implements Visitable {
2   public void accept(BinaryTreeVisitor visitor) {
3    visitor.visitNode(this);
4   }
5  }
6  ...
```

```
1  public class SummationVisitor implements
     BinaryTreeVisitor {
2   protected int sum = 0;
3
4   public void visitNode(Visitable node) {
5    BinaryTreeNode rnode = (BinaryTreeNode) node;
6    rnode.left.accept(this);
7    rnode.right.accept(this);
8   }
9
10  public void visitLeaf(Visitable node) {
11   BinaryTreeLeaf leaf = (BinaryTreeLeaf) node;
12   sum += leaf.getValue();
13  }
14  ...
15 }
```

Figure 99. Application of an visitor to visited classes.

4.23.2 Implementation

OOP solution. Hannemann et al. apply the visitor pattern to perform operations on a tree structure, like summation of tree elements (`SummationVisitor`) or to display the tree (`TraversalVisitor`). For that, the visitor object, e.g., of type `SummationVisitor`, is applied to the root node of the tree structure, using the `accept` method. This `accept` method invokes a `visitNode` or `visitLeaf` method of the overgiven visitor object using itself as parameter. Before or after processing the root node of the tree the visitor object applies itself to the children of the root node by invoking their `accept` method and thus the whole tree structure is traversed recursively before or after processing the nodes. This recursion end when a leaf is processed by the visitor (`visitLeaf`). The `visitLeaf` method processes the leaf node and returns. By backtracking the result for the root node, i.e., for the whole tree structure, is calculated, e.g., the sum of the elements. An example listing is depicted in Figure 99. The listing depicts the `accept` method of the class `BinaryTreeNode` (that objects are no leafs, Lines 1–6). If a visitor X is applied, its method `visitNode` is invoked (Line 3) giving the identity as parameter. The `visitNode` of the visitor object (Lines 10–14) processes a tree node, except leafs, i.e., it applies itself to the children of the tree node (Lines 12–13). `BinaryTreeLeaf` objects invoke the `visitLeaf` method of the visitor (Lines 16–19) and are processed directly, e.g., the value is added to the sum (Line 17).

Advantages The visitor classes `SummationVisitor` and `TraversalVisitor` each merge the code regarding one operation to be performed on the tree structure, e.g., summation of all elements of a tree. Consequently, new operations to be performed on the tree structure and can be added by adding further visitor classes.
The visited objects, e.g., of type `BinaryTreeLeaf` or `BinaryTreeNode`, do not have to implement the same interface nor do they have to be related at all. Hence, the visitor pattern allows to navigate across different class hierarchies.
The operation that can process unrelated objects of arbitrary types does not demand for global variables or additional parameters to keep track about the operation.

Disadvantages If objects of a new class, e.g., `AnotherTreeNode`, are introduced into the tree structure, all visitor classes, e.g.,

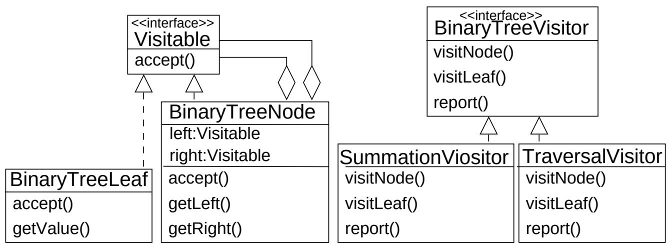

Figure 100. OOP implementation of the Visitor pattern.

`SummationVisitor`, have to be extended to process this new tree node type `AnotherTreeNode`.
The visitor, e.g., `SummationVisitor`, only can access public members of the visited classes `BinaryTreeNode` and `BinaryTreeLeaf` to perform its operation.
The extension by visitors has to be anticipated by implementing an `accept` method in every visitable class.

AOP solution. Hannemann et al. merge the code of associated to visitors of a tree structure (with two kinds of elements) into the aspect `VisitorProtocol`. The tree node classes, e.g., `BinaryTreeLeaf`, are assigned to implement the interfaces `Node` and `Leaf`. These interfaces implement the `accept` methods that allow visitor objects to process the tree node objects. Additionally, the aspect `VisitorProtocol` assigns the classes `SummationVisitor` and `TraversalVisitor` to implement the `Visitor` interface so that they can be applied to the `accept` methods of the tree elements.

Advantages The advantages of the OOP implementation hold for the AOP implementation.
The tree classes, e.g., `BinaryTreeNode`, do not have to be prepared to be processable for visitor objects, i.e., they do not have to implement the `accept` method from beginning.

Disadvantages Code regarding one class, e.g., `BinaryTreeNode`, is scattered across the classes `BinaryTreeNode`, `SummationVisitor`, and `TraversalVisitor`.
The visitor classes, e.g., `TraversalVisitor`, only can access public members of the tree node classes to perform their operation.

FOP solution. We present 2 approaches for that pattern: Solution A is close to the AOP implementation, see Figure 102. The tree node classes `BinaryTreeNode`, `BinaryTreeLeaf`, and `Visitable` are extended to inherit the method `accept` from the classes `Leaf` and `Node` respectively. That method allows visitors, like `SummationVisitor`, to process the tree node classes. Additionally, the visitor classes `SummationVisitor` and `TraversalVisitor` are assigned to the `Visitor` interface so that they can be applied to the `accept` method of the tree node classes.
Solution B is a simplification of solution A. Since the distinction of tree node types, e.g., `BinaryTreeNode`, in the visitor is implemented through distinction of method names in the `accept` method (see Fig. 99) the classes `Leaf` and `Node` can be omitted (Fig. 103). Mixin classes that introduce the `accept` methods inherit the `VisitableNode` interface left. The methods defined in the classes `Node` and `Leaf` are transfered into refinements of the tree node classes.

Advantages The advantages of the OOP and AOP implementation hold for the FOP implementation.

Criteria	OOP	AOP	FOP
Cohesion	0	+	+
Variability	0	+	+

Table 23. Evaluation of the pattern Visitor

Disadvantages Visitors only can access public members of the tree node classes, e.g., `BinaryTreeNode`, to perform their operation.

4.23.3 Discussion

Cohesion. We haveto consider two issues:

- In the OOP, AOP, and FOP implementations the code associated to a tree node class, e.g., `BinaryTreeLeaf`, is scattered across the respective class, e.g., `BinaryTreeLeaf`, and the visitors `SummationVisitor` and `TraversalVisitor`.
  In the OOP, AOP and FOP implementations the code regarding one operation is merged in the visitor classes, e.g., `SummationVisitor`.
- In the OOP implementation the tree node classes, e.g., `BinaryTreeNode`, are closely coupled to variant behavior of operations due to the `accept` method.
  In the AOP and FOP implementations the tree node classes are not coupled to the variant behavior of the visitor classes, because the `accept` method is introduced subsequently.

Variability. In the OOP implementation the visitor classes, e.g., `SummationVisitor`, are restricted to the classes that are subtype a common interface `Visitor` so that they can be accepted by the tree nodes. In the AOP and FOP implementations all classes, that provide the `visitNode` and `visitLeaf` method can be used as a visitor due to subsequent extension.

4.23.4 Summary

The primary problem the visitor pattern aims to solve is to extend classes without changing them – that can be surfed by simple AOP introductions or FOP refinements respectively.
Since Java prevent multiple inheritance and in that AOP implementation indeed multiple inheritance is used the transformed feature module in FOP has to be split. Furthermore, the `VisitableNode.accept` method had to be deleted, which does not matter because the only class inheriting this implementation overrides this method anyway.
The authors criticize that classes inherit from internal classes of an aspect directly.

5. Conclusions

GoF design patterns are well known and used to improve flexibility and reusability of software that is implemented in OOP. Hannemann et al. observed a lack of modularity in object-oriented design pattern implementations and thus improved the pattern implementations using AOP [15]. We followed the line of Hannemann et al. and reimplemented their design pattern implementations with FOP. We defined criteria *cohesion* and *variability* and used these criteria to evaluate and compare OOP, AOP, and FOP design pattern implementations.

Acknowledgments

The authors want to thank Sven Apel, Marko Rosenmüller, and Roberto E. Lopez-Herrejon for helpful discussions and remarks.

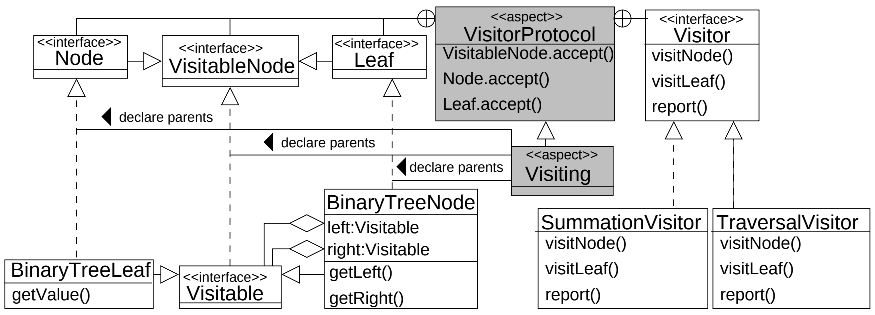

Figure 101. AOP implementation of the Visitor pattern.

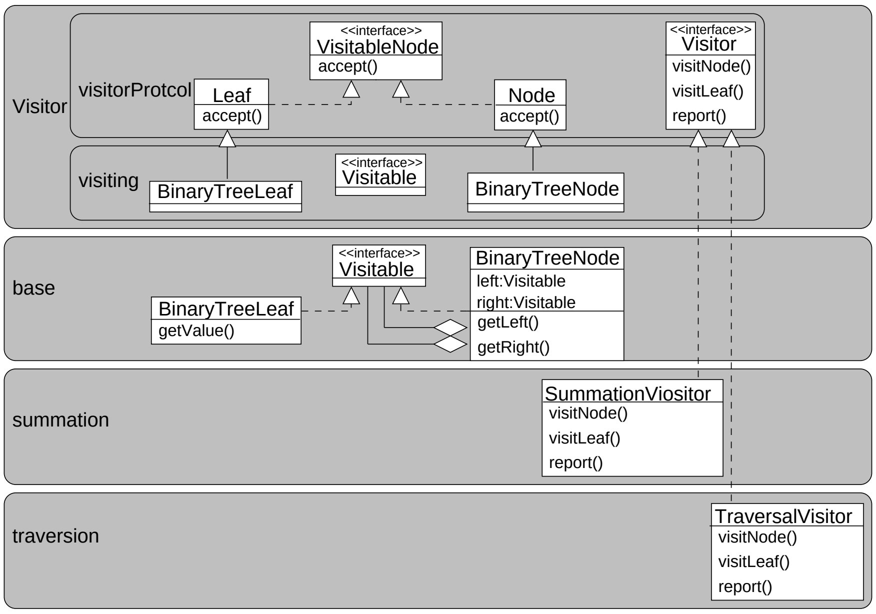

Figure 102. FOP implementation of the Visitor pattern.

References

[1] T. Aotani and H. Masuhara. Compiling Conditional Pointcuts for User-Level Semantic Pointcuts. In *Software Engineering Properties of Languages and Aspect Technologies*, 2005.

[2] S. Apel and D. Batory. When to Use Features and Aspects?: A Case Study. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 59 – 68, 2006.

[3] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 122–131, 2006.

[4] D. Batory, J. Liu, and J. N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 48–57, 2003.

[5] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating Product-Lines of Product-Families. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, page 81, 2002.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[7] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[8] A. Charfi, M. Riveill, M. Blay-Fornarino, and A.-M. Pinna-Dery. Transparent and Dynamic Aspect Composition. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2006.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] K. Driesen and U. Hölzle. The Direct Cost Of Virtual Function Calls in C++. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 306–323, 1996.

[11] U. W. Eisenecker, F. Blinn, and K. Czarnecki. A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In *Workshop on C++ Template Programming*, 2000.

[12] E. Ernst. Syntax Based Modularization: Invasive or Not? In *Workshop on Advanced Separation of Concerns*, 2000.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. v. Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the International Conference*

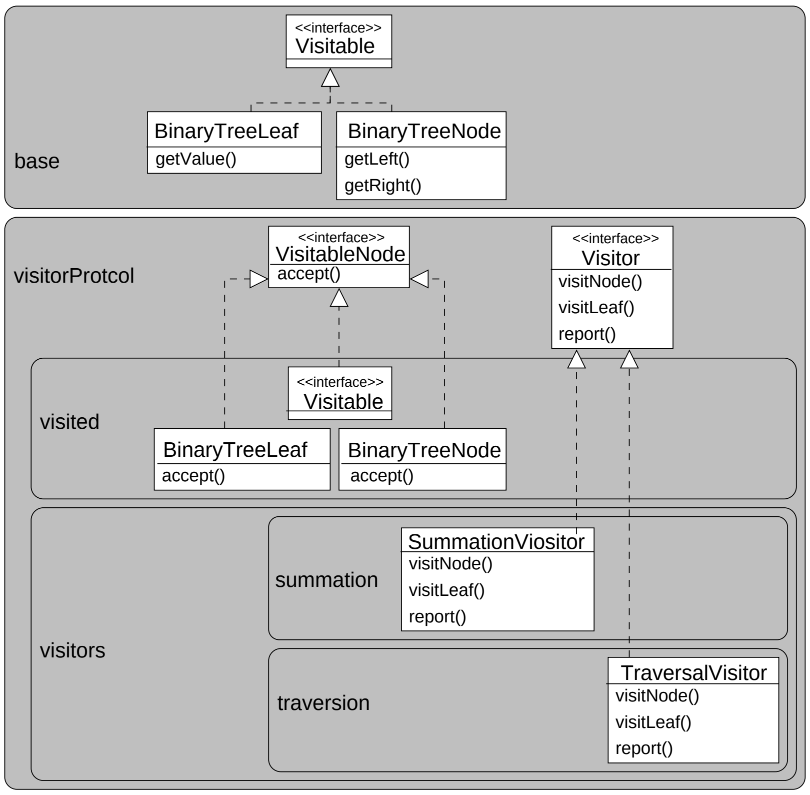

Figure 103. Alternative FOP implementation of the Visitor pattern.

on Aspect-Oriented Software Development (AOSD), pages 3–14, 2005.

[15] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, 2002.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[17] R. Lopez-Herrejon, D. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 169–194, 2005.

[18] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 46–60, 2003.

[19] B. Meyer. *Object-oriented software construction*. Prentice Hall PTR, 2 edition, 1997.

[20] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90 – 99, 2003.

[21] T. Millstein and C. Chambers. Modular Statically Typed Multimethods. *Lecture Notes in Computer Science*, 1628:279 – 303, 1999.

[22] Object Management Group. *OMG Unified Modeling Language Specification (Version 1.4.2)*, 2005.

[23] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 734–737, 2000.

[24] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443, 1997.

[25] A. Rashid, P. Sawyer, A. M. D. Moreira, and J. Araújo. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In *Proceedings of the International Conference on Requirements Engineering (RE)*, pages 199 – 202, 2002.

[26] Y. Smaragdakis and D. S. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570, 1998.

[27] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 107–119, 1999.

[28] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOPS Messenger*, 1(1):7–87, 1990.