Universitatea Politehnica din Timişoara
Facultatea de Automatică şi Calculatoare
Departamentul Calculatoare

Automation of Complex Design Flaws Detection

Károly Szántó

Proiect de Diplomă

Iunie 2009

Conducător Ştiinţific:
Dr. ing. Adrian Trifu

Any fool can make things bigger, more complex, and more violent.
It takes a touch of genius – and a lot of courage – to move in the opposite direction.

Albert Einstein (1879 - 1955)

Contents

Chapter 1

Introduction

1.1 Context of the Work

In [Pre00], *software design* is defined as an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of the software. That is, the design is represented at a high level of abstraction in the form of an abstract model which is to be materialized in source code. The set of all entities, their properties, and relations, which can be extracted from the source code of the system, using ordinary parsing and type analysis techniques represents the *static structure*[1] of the system.

As defined in the ANSI/IEEE standard 729-1983, *software maintenance* represents "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment". Because maintenance is the most expensive phase in the life cycle of a complex software system, there is a motivated need for the existence of methods that ensure:

- a high quality of the initial design, in order to minimize the need and extent of subsequent maintenance activities;
- a low cost of the maintenance activities themselves.

If we refer to the software design from the context of maintainability, the quality of the design is measured from the point of view of the maintainer: the ease with which he can understand and accommodate changes to the design of the system. Maintainers often have to rely on the source code alone in order to understand the design, because the documentation is either outdated or of little practical use. That's why the *the structure of the source code* is of high importance in the maintenance process.

The main task of the static structure should be to keep maintenance effort as low as possible. As a result, any characteristic of a fragment of the structure, that has a negative impact on maintenance effort, constitutes a *structural anomaly* [Ciu01].

[1]From now on will be referred simply as "the structure"

As indicated in [Tri08], a bad quality of the static structure has a negative impact on the maintainability of a software system, in the following two ways:

1. an "unnatural" solution structure in the context of the design problem at hand affects the maintainer's ability to understand the design;
2. since any given structure favors specific types of change while hindering others [Mar03], a solution structure that does not take into account certain variability points mandated by the design problem at hand, may affect the maintainer's ability to operate changes in response to changing requirements. In other words, in order for the design to be easily changeable, the designer must take into account the types of changes that are more probable to occur in the future.

Repetitive changes to the design of a system will favor the apparition of the phenomena described above, leading to the degradation of the static structure. In other words, as a software system evolves, it becomes more complex, and extra resources are needed in order to preserve and simplify its structure. This happens because repeated changes tend to degrade the system's structure, a phenomenon referred to as "software aging" [Par94].

In order for a system to fight against software aging, its source code structure needs to be periodically improved, without changing the system's behaviour; this is called the *restructuring process.*

In [Bae99] there are three steps identified for a restructuring process, based on structural anomalies:

1. *Problem detection*: is concerned with finding instances of structural anomalies in the subject system.
2. *Problem analysis*: covers the activities that are involved in deciding how to improve the structure of the design fragment under consideration.
3. *Reorganization*: deals with the implementation of the new structure, decided upon in the previous step.

1.2 Problem Statement

Two of the steps listed above, problem detection and reorganization, can be solved in an almost fully automated manner, but the problem analysis step is a mostly manual process that requires experience and intuition from the maintainer's side.

Structural anomalies, also called "code smells", can be detected automatically and they can be used as a starting point in the analysis process. But a code smell can warn us about the presence of a multitude of different design deficiencies. This is the reason for the symptomatic nature of the structural anomalies and that is why it is hard to define a reorganization strategy which can be uniquely associated to an anomaly.

All the reasons described above confer a costly nature to the whole restructuring process. As a result, there is a compelling amount of pressure placed upon the research community, to come up with methods that ensure the automation of the three steps, which characterize the restructuring process, described above.

1.3 Goal

The long term goal of our research is to increase the automation as much as possible in the restructuring process.

An important first step has been done in [Tri08]: transformation of the restructuring process, from a process that is heavily dependent on intuition and personal know-how, into a systematic process that supports automation. The approach was based on the simple idea of constructing higher-level entities, called *design flaws*, as triplets that consist of a *design context*[2], a *pathological structure*[3] and a *reference structure*[4].

1.4 Overview of Contributions

Our work is based and builds upon the theoretical foundations and practical results of [Tri08]. As a result, our contributions were structured as follows:

Theoretical contributions:

1. Improve the definition of five design flaws: *Collapsed Type Hierarchy*, *Embedded Strategy*, *Explicit State Checks*, *Dispersed Control* and *Embedded Features.* The improvements were based on the experimental results from [Tri08].
2. Define new design flaws in order to extend the catalog.

Practical contributions:

- Implement three new design flaws into *CodeClinic*[5]: *Embedded Strategy*, *Dispersed Control* and *Embedded Features.*
- Improve the detection mechanism in the current implementation by accomodating the changes that occurred to the design flaws definitions.

[2]consists of two elements: design intent and strategic closure
[3]illustrates an anti-pattern in the given design context
[4]represents a possible solution in the given design context
[5]The prototype tool

- Improve CodeClinic's framework by implementing the support for *initial filter*[6] and better structures to represent the elements affected by design flaws.

1.5 Structure of the Thesis

The rest of the work is structured as follows: Chapter 2 covers the terminology use in the current work and provides an overview of CodeClinic. Chapter 3 reviews and compares other approaches, highlighting their deficiencies with respect to the criteria defined in the current work.

Chapter 4, 5 and appendix A constitute the core of the thesis. Chapter 4 presents the theoretical contributions brought by the current work while Chapter 5 discusses the implementation details of the practical work. Finally, appendix A contains the catalogue of redefined design flaws, being ready to be used in day to day practice.

Chapter 6 is dedicated to the evaluation of CodeClinic and validation of the changes that occurred in the design flaw definitions by comparing the evaluation results with the results from [Tri08]. Chapter 7 summarizes the contributions brought by the current thesis and presents the main goals of our future work.

[6]Captures features that are mandatory for any instance of a design flaw

Chapter 2

Background

The theoretical part of the current work is based on [Tri08] and the practical part is represented by improvements and added features to the prototype tool, *CodeClinic*. That is why, in order to describe the accomplishments achieved in the current work, the understanding of some terms defined in the above mentioned work is essential. It is also important to shortly describe the prototype tool as it was when the current work started and the approach taken in order to transform the restructuring process, from a process that is heavily dependent on intuition and personal know-how, into a systematic process that supports automation.

2.1 Terminology

The definitions bellow are as stated in [Tri08] unless otherwise specified.

Design Context: The *design intent* and the *strategic closure* corresponding to a design fragment will collectively be referred to as the *design context* of that fragment.

Design Flaw: A triplet that consists of a *design context*, a *pathological structure* and a *reference structure.*

Design Intent: is an abstract description of what needs to be achieved, in a given fragment of design.

Initial Filter: Definition of constraints for a design flaw to reduce the search space for the analysis as much as possible. A valid candidate will be one that passes the initial filter and that has at least one valid indicator.

Pathological Structure: Illustrates an anti-pattern in the given design context.

Strategic Closure: OCP[1] states that modules should be closed for modification but open for extension. "In general, no matter how "closed" a module is, there will always

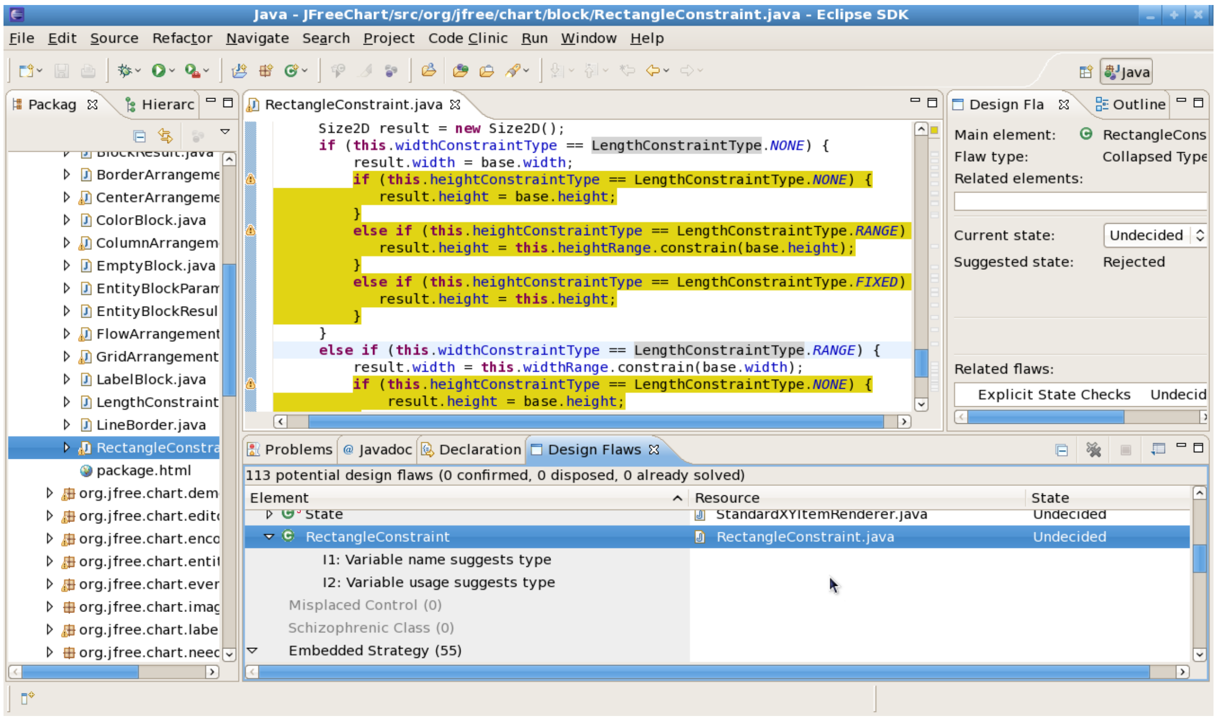
[1]Open/Closed Principle

be some kind of change against which it is not closed. There is no model that is natural to all contexts!" [Mar03]

The types of changes that have a higher probability will be referred to as the *strategic closure*.

2.2 CodeClinic

CodeClinic was created as an Eclipse[2] plug-in. This is illustrated in figure 2.1. The tool makes it possible for the developers to scan a project, a package or even just a single compilation unit for potential design flaws. The detection is carried out in the background. The candidates found during the detection are presented in two views:



```
    Size2D result = new Size2D();
    if (this.widthConstraintType == LengthConstraintType.NONE) {
        result.width = base.width;
        if (this.heightConstraintType == LengthConstraintType.NONE) {
           result.height = base.height;
        }
        else if (this.heightConstraintType == LengthConstraintType.RANGE)
           result.height = this.heightRange.constrain(base.height);
        }
        else if (this.heightConstraintType == LengthConstraintType.FIXED)
           result.height = this.height;
        }
    }
    else if (this.widthConstraintType == LengthConstraintType.RANGE) {
        result.width = this.widthRange.constrain(base.width);
        if (this.heightConstraintType == LengthConstraintType.NONE) {
            result.height = base.height;
```

Main element: RectangleCons
Flaw type: Collapsed Type
Related elements:
Current state: Undecided
Suggested state: Rejected
Related flaws:
Explicit State Checks Undecid

113 potential design flaws (0 confirmed, 0 disposed, 0 already solved)

Element	Resource	State
State	StandardXYItemRenderer.java	Undecided
RectangleConstraint	RectangleConstraint.java	Undecided
I1: Variable name suggests type		
I2: Variable usage suggests type		
Misplaced Control (0)		
Schizophrenic Class (0)		
Embedded Strategy (55)		

Figure 2.1: Screen shot of CodeClinic

Design Flaws: In this view are listed all the main elements of each detected design flaw and each element can be expanded to consult its indicators. By double-clicking on an element it will be opened in the environment's main editor window where the affected code fragments are highlighted.

[2]http://www.eclipse.org/

Design Flaw Details: When an element is selected in the *design flaws* view the following details will be presented in this view:

- The main element of the design flaw.
- The type of the design flaw.
- Related elements. This is needed because a design flaw can affect more than one element, so the developer can easily jump from one affected element to another.
- The design flaw's current state. This can be changed at anytime either from this view or by right-clicking an element in the *design flaws window* and selecting a new state. A design flaw candidate can be in one the following states: "Confirmed", "Rejected", "Undecided" and "Solved".
- Related design flaws. This is important because an element can be affected by more than one design flaw.

When the current work started, there were five design flaws implemented in the prototype tool:

- Containment By Inheritance
- Explicit State Checks
- Collapsed Type Hierarchy
- Misplaced Control
- Schizophrenic Class

In order to see the definition of the above mentioned design flaws as they were before the current work, please consult Appendix A in [Tri08].

Chapter 3

Related Work

The previous chapter is dedicated to the introduction of basic terminology, as well as the description of the foundations upon which the current work is built. In this chapter we will describe some tools that aim at the problem at hand, that is software aging, by contributing to the automation of the restructuring process described in chapter 1.

3.1 JDeodorant

JDeodorant[1] is an Eclipse plug-in that automatically identifies, by employing some novel methodologies and resolves design problems in software, known as bad smells. For the moment, the tool identifies two kinds of bad smells, namely "Feature Envy" and "Type Checking" bad smells. "Feature Envy" problems are automatically resolved by "Move Method" and "Extract and Move Method" refactorings. "Type Checking" problems are automatically resolved by "Replace Conditional with Polymorphism" and "Replace Type code with State/Strategy" refactorings.

The tool is the outcome of the research effort in the Computational Systems and Software Engineering Lab, at the Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. JDeodorant encompasses a number of innovative features:

- Transformation of expert knowledge to fully automated processes.
- Pre-Evaluation of the effect for each suggested solution.
- User guidance in comprehending the design problems.
- User friendliness (one-click approach in improving design quality).

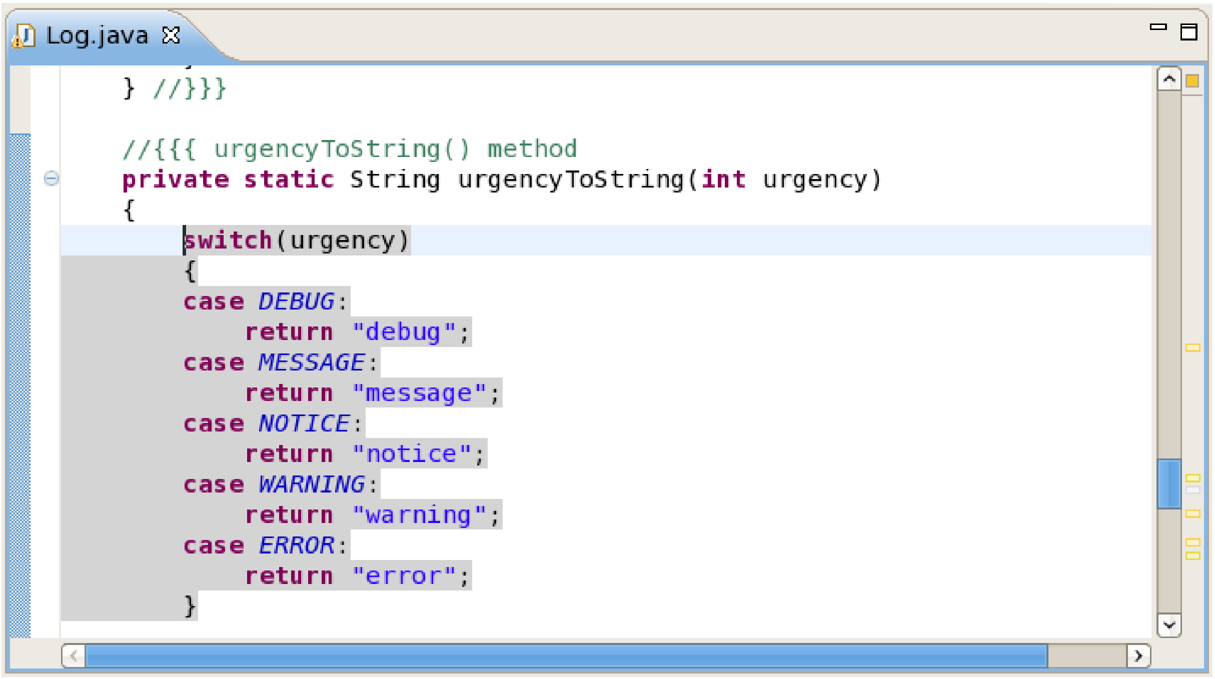
As mentioned above, "Type Checking" is one of the two bad smells the tool identifies at the moment. This kind of smell targets the conditional constructs and can be resolved by the means of two different refactorings, depending on the structure of the conditional expressions:

[1]http://www.jdeodorant.com/

- If the expression checked by the conditional construct verifies the concrete type of a variable, the "Replace Conditional with Polymorphism" refactoring will be invoked.
- If the expression checks a variable against some symbolic constants, the "Replace Type code with State/Strategy".

The described approach is simple, but there are no semantics taken into account in the identification process.

For example, figure 3.1 depicts an example of "Type Checking".



```
    } //}}}

    //{{{ urgencyToString() method
    private static String urgencyToString(int urgency)
    {
        switch(urgency)
        {
        case DEBUG:
            return "debug";
        case MESSAGE:
            return "message";
        case NOTICE:
            return "notice";
        case WARNING:
            return "warning";
        case ERROR:
            return "error";
        }
```

Figure 3.1: Bed smell detected by jDeodorant in the JEdit project

As can be seen, the conditional construct returns a different string for each value the checked variable can have. jDeodorant suggests that this conditional should be refactored by the means of "Replace Type code with State/Strategy" without having any knowledge neither about the nature of the constants the variable is checked against, nor the nature of the code inside the branches of the conditional. In this case the symbolic constants represent just some urgency levels and the presented method determines their specific string representation.

The point is that, with this approach, a lot of false positives are detected and a problem cannot be put into a direct correlation with a unique solution. For instance, if we have a conditional construct, like in the above example, but the symbolic constants do not represent states of the affected class and the code inside the conditional's branches does not represent an algorithm, then again, the refactoring does not make any sense.

All together, this tool, based on simple approach, can detect simple design problems based on code smells and, if needed, refactor the affected code by the means of the above mentioned refactorings.

3.2 inCode

inCode[2] is an Eclipse plug-in that helps in understanding, assessing and improving the quality of the projects' design.

The main features of the tool, that are relevant in the context of the present work, are:

- Detection of well-known "design flaws"[3]: Code Duplication, Data Class, God Class and Feature Envy. The mechanism behind the detection of design flaws is based on object-oriented metrics.
- Smart contextual advice for the detected design problems. Besides that, customized advices available on how to refactor the affected code in order to improve the quality of the analysed object-oriented software system.

Taking a look at the description of the tool, a first impression would be that it resolves the same problems we have defined in chapter 1 by detecting "design flaw" instances and giving refactoring hints for them. The difference is that for *inCode*, "design flaw" is just a synonym for "code smell", while in our work *design flaw*, as already mentioned, is defined as a higher-level entity that is specified by three components. One of these components is the *design context* which gives semantics to the problem by specifying an abstract description of what needs to be achieved, in a given fragment of design, and by taking into account the types of changes that have a higher probability to occur in that fragment of design.

So, the main difference between the approach of the current work and the approach of *inCode* can be found in the definitions of the terms that they build upon.

The second difference is that *inCode* measures the analysed systems in the terms of object-oriented metrics, as defined in [Mar01], and gives different feedback by interpreting them in the context of that system, while we try to detect complex design problems and put them in concordance with a unique pattern based refactoring solution.

[2]http://www.intooitus.com/inCode.html
[3]They refer to code smells in the context of inCode

3.3 Conclusions

This chapter gave an overview of research that is aimed at the problem at hand, by automating the *problem detection* step of the restructuring process. The detection mechanisms of the above mentioned tools are simply based on code smells. Therefore, they might have automate the *problem detection* step and the *reorganization* step, but the *problem analysis* step still remains a mostly manual process, based on intuition and personal knowledge

We conclude that because of the approach that tries to bridge the gap that currently exists, between quality assessment of software structure and code transformation, CodeClinic is an evolutionary tool that brings automation to the *problem analysis step*.

Chapter 4

Contributions to the Existing Catalogue of Design Flaws

As pointed out in section 1.3, the theoretical contributions brought by the current work are:

1. Redefine some design flaw definitions.
2. Definition of new design flaws.

4.1 Redefined Design Flaws

The first theoretical goal to be achieved in the current work was to improve the definition of *Collapsed Type Hierarchy* [A.1] and *Explicit State Checks* [A.3] design flaws.

After analysing the false positives detected by the tool, the above mentioned design flaws were redefined and, after modifying the source code to be in concordance with the new definitions, their improvement was proven in 6.

Further on, after implementing and evaluating *Embedded Strategy* and *Embedded Features* as specified in [Tri08], they also got redefined.

4.1.1 Collapsed Type Hierarchy

A collapsed type hierarchy is the situation where an abstraction "absorbs" its own specializations, and emulates the specialization hierarchy, by explicitly checking the value assigned to a variable that represents the object's special type. The checked variable will simply be referred to as the *selector*.

The object's special type represents a characteristic of the affected class, therefore both the variable which represents the object's type and the constants that represent the emulated hierarchy's types belong into that class. As a result, the natural approach to represent a type of the emulated hierarchy would be to store it into one of the class' attributes. The hierarchy's types should be represented as symbolic constants defined as *final* attributes of the affected class or as constants of an enum.

When the instance of a class is created, its concrete type is specified at the moment of creation so, the object's special type should be controlled only be the clients of the affected class. Therefore, The above mentioned class attribute, which holds the object's special type, has to be modified only in the constructor of the class, in the a setter or directly by the client.

In the old definition, the *initial filter* was too permissive in the sense that it would let pass any class containing at least two conditional constructs that check upon a selector which is a class attribute or formal parameter. As stated above, it is unnatural for a client to call a method of the affected class with a parameter representing an object type. The methods should be invoked on an object which has a concrete type not on an object who's type is specified when a method is invoked on it.

The fact that a formal parameter could be considered as a selector, introduced a substantial number of false positives.

Another aspect of the initial filter was that it did not take into account the structure of the conditional expressions checked by the affected conditional constructs. In order to emulate the behaviour of a special type, the attribute which stores the object's type should be checked against one of the symbolic constants which represent the hierarchy's types and based on this, specific operations would be performed for each concrete type.

If the conditional construct is a "SWITCH" we don't need to worry about the conditional expression, but if the conditional construct is an "IF- ELSE IF ..." structure the initial filter should only take into account the ones that have infix conditional expressions which have a structure similar to **selector == symbolic_constant** or **getSelector() == symbolic_constant**.

Although the statistics had shown that the definition of the first indicator was not really good, we left it unchanged because it makes perfect sense that the names of the variables, which represent types of the emulated hierarchy, contain the string "type".

The second indicator specified that usage patterns of the selector used in the conditionals suggest a type variable. In other words, there is no write access on the variable, anywhere in the class, with the exception of object constructors. As we already said, the object's special type should be controlled from outside the class, by the affected class' clients. Thus, the attribute representing the object's type could also be modified through its setter or directly by one of the classes clients. As a result, this statement became part of the second indicator.

In listing 4.1 is presented a relevant sequence from a class that is affected by the *Collapsed Type Hierarchy* design flaw. It is clear that, as discussed above, the type variable is represented by a class attribute and it is compared against symbolic constants, representing

```
...

/** The constraint type. */
private LengthConstraintType heightConstraintType;

...

public Size2D calculateConstrainedSize(...) {
  ...
  if (this.heightConstraintType == LengthConstraintType.NONE) {
    result.height = base.height;
  }
  else if (this.heightConstraintType == LengthConstraintType.RANGE) {
    result.height = this.heightRange.constrain(base.height);
  }
  else if (this.heightConstraintType == LengthConstraintType.FIXED) {
    result.height = this.height;
  }

  ...
}
}
```

Listing 4.1: Collapsed type hierarchy in class `RectangleConstraint` in `JFreeChart` project

classes of the emulated hierarchy, defined as *static* and *final* attribute in the *LengthConstratintType* class. More than that, the conditional expressions checked in the conditional construct are infix expressions and they clearly have the above specified structure.

4.1.2 Explicit State Checks

Explicit state checks refers to the situation in which an object uses explicit checks on some internal piece of data, in order to execute state specific behavior or manage its “state” transitions. The data that is checked represents the current “state” at any given time.

As a result, the data which represent the current state of the object belongs into the affected class. So, the states of the affected class could be managed in one of the following two ways:

- The possible states of the class are symbolic constants represented as *final* attributes of the class or as constants of an enum which are assigned to one of the class’ attributes ,that represents the “state” of the object, in order to perform transitions between the states. The attribute is checked in simple conditional constructs against the symbolic constants that represent possible states of the object in order to perform state specific behaviour.

- The state of an object is represented by the values of all its attributes at a moment. Thus, the state of the affected class could be represented by the values of a group of its attributes which are checked always checked together in the affected conditionals of the class in order to perform state specific behaviour.

Concluding the above stated methods, the conditional expressions that are checked in an "IF - ELSE IF ..." construct should have a structure similar to **selector == symbolic_constant** or **class_attribute_1 == value_1 boolean_operand ... class_attribute_n == value_n**.

So, the initial filter was modified in order to accommodate the facts stated above: it will let pass only classes that contain at least one attribute that represents a state variable. That is, the attribute is checked, in simple conditional constructs which have a structure similar to the ones described above, against symbolic constants representing the possible states of the affected class.

The values of the attributes representing the state of the object should be managed by both the client and the affected class. The client specifies the initial state of the object or sets a specific state at any given time, while the affected class manages the transitions of its state inside the branches of the conditionals that perform state specific behaviour.

Although the statistics had shown that the definition of the first indicator was not really good, we left it unchanged because it makes perfect sense that the names of the variables, which represent states of the affected class, contain the string "state".

The second indicator specified that usage patterns of the selector used in the conditionals suggest a state variable, in the sense that the value of the checked parameter is changed, either within branches of the conditional constructs, or from the clients that called the respective operation. The fact that the state variable could be changed by the clients of the affected class was made a little more explicit: the client of class could specify the initial state of the object through the object's constructor or change its state through a setter.

In listing 4.2 is presented the affected code from a method inside a class affected by this design flaw. The state variable *xPosition* is defined as attribute of the affected class and the states are represented by symbolic constants defined as *static* and *final* attributes in the *TimePeriodAnchor* class. More than that, the state variable is given an initial state in the constructor and it's state is modified by clients of the class through the selector's setter. All the arguments above makes the example in the listing a clear candidate of the discussed design flaw.

```
  ...

  private TimePeriodAnchor xPosition;

  ...

  private long getX(TimePeriod period) {
    ...
    if (this.xPosition == TimePeriodAnchor.START) {
      return period.getStart().getTime();
    }
    else if (this.xPosition == TimePeriodAnchor.MIDDLE) {
      return period.getStart().getTime()
          / 2 + period.getEnd().getTime() / 2;
    }
    else if (this.xPosition == TimePeriodAnchor.END) {
      return period.getEnd().getTime();
    }
    else {
      throw new IllegalStateException("TimePeriodAnchor unknown.");
    }

    ...
  }
  }
```

Listing 4.2: Explicit state checks in class `TimePeriodValuesCollection` in `JFreeChart` project

4.1.3 Embedded Strategy

An embedded strategy refers to the situation in which the class providing the context, explicitly switches between alternative algorithms, whose implementations are all hard-coded into the class itself.

Like in the other already described design flaw definitions, experiments showed that the *initial filter* is too permissive. It would let pass any class which has at least one conditional construct that checks upon a class attribute or formal parameter. The problem here is that class attributes are taken into account. Based on evaluation results, it was obvious that the common case for this kind of flaw is to parameterize a method with a value which will be checked in order to select the corresponding “strategy”, then to parameterize the whole affected class by the means of one of it’s attributes. As a matter of fact, the whole idea behind this design flaw is that the strategy to be applied in a specific situation depends on the values passed by the client to the affected method. That’s why now only formal parameters are considered as possible parameters for the selection of a specific strategy.

The definition of the first indicator, like in the case of “collapsed type hierarchy” and “explicit state checks”, looks for possible problems in the names of the parameters used to select a specific strategy by verifying if the name of parameters contain one of the strings “strategy” or “algorithm”. This definition seemed quite reasonable so it was left unchanged.

We wanted to make the *initial filter* more restrictive therefore, the second indicator was made part of the initial filter. We chose this indicator because in its definition it is specified that the parameters checked upon in the conditional constructs in order to select a specific strategy should not be modified inside the containing method. It is obvious that such a parameter should only be used as an input parameter.

As stated above, the implementations of all strategies are hard-coded into the affected class itself. These algorithms are represented by the branches of the conditional constructs. Therefore, we need to define the meaning of an “algorithm” in the given context. We came up with a definition which states that a code sequence that represents a hard-coded strategy is an “algorithm” if all its method calls are to private methods of the affected class, except the getters and the setters. This seems to be correct because if the strategy is hard-coded in the class, the pieces that make it up should be hidden parts of that class.

With the above definition at hand, we have redefined the second indicator: the body of each branch represents an algorithm, method calls should be to private methods of the class. That is, all method calls should be to private methods of the containing class, except the getters and setters.

The definition of indicator 3 points out that the concern being specialized represents a small fraction of the class. In other words, a very small number of the class' non-accessor methods contain the affected conditional constructs. If the same strategy is selected in more than one method of the class its algorithm should be the same. As a result, we have enhanced the definition of this indicator by adding the following constraint: if the conditional construct is contained in more that one method, there should be signs of code duplication between the bodies of the same branches.

This latest change in the third indicator was not implemented in CodeClinic because, at the moment, we do not have support for the detection of code duplication.

Listing 4.3 illustrates a method affected by the embedded strategy design flaw. As shown, *menuItem* is used as a selector in the conditional construct and it's value is not modified in the method (only read accesses are performed on it). If more types of graphical components will become posible menu items, it would be motivated to refactor the code in the listing by the means of restructuring defined in A.2.

```
private JMenuItem addMenuItem(..., int menuType, ...) {
    JMenuItem menuItem;
    if (menuType == 0)
      menuItem = new JMenuItem(menuTexts);
    else if (menuType == 1) {
      menuItem = new JCheckBoxMenuItem(menuText);
      ((JCheckBoxMenuItem) menuItem).addItemListener(this);
    }
    else {
      menuItem = new JRadioButtonMenuItem(menuText);
      ((JRadioButtonMenuItem) menuItem).addItemListener(this);
    }
    ...
}
```

Listing 4.3: Embedded strategy in class `XEditorMenu` in `XUI` project

4.1.4 Dispersed Control

We have a case of "dispersed control", when in a situation such as the one described in the first part of A.4's definition, the implementations of structure related functionalities are broken up and dispersed between the individual types that belong to the structure. This design flaw affects type hierarchies.

The reference structure of this flaw implies the use of Visitor pattern [GHJV96]. Because "Most of the time you don't need Visitor, but when you do need Visitor, you really need Visitor!" [Ker04], the diagnosis strategy of this design flaw has to be very accurate. This was not the case for the old definition. After implementing it, in the evaluation process, we have noticed that in a medium-sized project (˜ 80000 LOC[1]), this design flaw has one or two valid candidates, and there are cases it has none. As a result, the initial filter has to be very restrictive in order to filter out as much false positives as possible.

First, we have decided that only hierarchies which have at least one client should be taken into consideration. This decision was based on the fact that the problematic nature of the hierarchy, as stated in this design flaw's definition, depends on how the clients make use of its classes and interfaces.

The experiments have shown that the first indicator was present in almost all detected instances, so we have decided to move its definition into the initial filter.

After the above mentioned modifications, the initial filter now has the following mandatory conditions for a hierarchy to be considered a possible candidate:

1. The hierarchy should have at least one client.

[1]Lines of Code

```
public interface CategoryAnnotation {
    public void draw(Graphics2D g2, CategoryPlot plot,
             Rectangle2D dataArea, CategoryAxis domainAxis,
             ValueAxis rangeAxis);
}
```

Listing 4.4: Root of hierarchy

```
public class CategoryLineAnnotation implements CategoryAnnotation...{
  ...
  public void draw(Graphics2D g2, CategoryPlot plot,
           Rectangle2D dataArea, CategoryAxis domainAxis,
           ValueAxis rangeAxis) { ... }
  ...
}
```

Listing 4.5: Leaf of hierarchy

2. The root of the hierarchy defines a number of methods, either concrete or abstract, that are either overridden or implemented in almost all terminal nodes of the hierarchy.
3. The overridden methods represent bits of semantically unrelated global operations. Thus, there are no calls between any pair of such methods.

The definitions of indicator 2 and indicator 3 are both about the way the clients of the affected hierarchy use the methods of its classes and interface. From the experimental results we have inferred that the third indicator has a really good definition because it was present in all the confirmed instances. The second indicator was also present in all the confirmed instances, but it was also present in a lot of false positives. Therefore, we concluded that indicator 2 in not really relevant when it is the only indicator present, so we merged the definition of the second and third indicator into a single indicator.

After the above described redefinition, this design flaw has an initial filter and one indicator. A possible candidate, hierarchy which passes the initial filter, is validated only if the indicator is also present. As a conclusion, a hierarchy becomes a candidate only if it fulfills all the conditions mandated by both the initial filter and the indicator.

A clear instance of this design flaw is depicted in listings 4.4, 4.5 and 4.6. This is a hierarchy from the *JFreeChart* project used in the evaluation process.

4.1.5 Embedded Features

Embedded features is the situation when a class uses attributes that represent on/off switches for optional features of the class. The attributes are explicitly checked in order

```
public class CategoryPlot ... {
  ...
  protected void drawAnnotations(Graphics2D g2, Rectangle2D dataArea) {

      if (getAnnotations() != null) {
          Iterator iterator = getAnnotations().iterator();
          while (iterator.hasNext()) {
            CategoryAnnotation annotation = (CategoryAnnotation)
                 iterator.next();
            annotation.draw(g2, this, dataArea, getDomainAxis(),
                 getRangeAxis());
          }
      }
    }
  ...
}
```

Listing 4.6: Client

to choose the desired behavior in each case A.5.

In the old definition of the *initial filter* specifies the following conditions that a class have to fulfill in order be considered a potential candidate:

1. Defines at least one attribute that appears to represent an optional feature of the class. In other words, the attribute has a boolean type, and it is written to either in a constructor, a method whose name contains "initialize", "setup" or "configure", or from outside the class (either directly or through accessors), but not from other non-accessor methods of the class.
2. The attribute is checked exclusively in simple conditional statements (if or if-else), in methods of the class.

After the evaluation, it turned out that some possible candidates were not taken into consideration because the attributes of some classes that would fulfill the above mentioned conditions were initialized in methods containing the "update" string in their names. As a result, the initial filter now takes into account these kind of attributes.

The condition for the attribute type to be primitive was also changed and made less permissive: the attribute type has to be boolean. This is because, as the experiments have shown, an "optional" feature would be represented by a *boolean* attribute and not by an *int* or *char* or any other primitive type. That is because it is closer to the natural language to write, for instance: *hasBorder = true / false*.

Because it would of no use to refactor a class that has only one optional feature, one more condition was added to the initial filter: the class must have at least two different such attributes (attributes that represent optional features).

```
    ...
      /**
       * A flag that controls whether or not the line is visible - see
           also
       * shapeVisible.
       */
      private boolean lineVisible;
      /**
       * A flag that controls whether or not the shape is visible - see
           also
       * lineVisible.
       */
      private boolean shapeVisible;
      /** A flag that controls whether or not the shape is filled. */
      private boolean shapeFilled;
      /** A flag that controls whether or not the shape outline is visible
          . */
      private boolean shapeOutlineVisible;
    ...
  public void draw(Graphics2D g2, Rectangle2D area) {
    ...
    if (this.lineVisible) {...}

    if (this.shapeVisible) {
        ...
        if (this.shapeFilled) {...}
        if (this.shapeOutlineVisible) {...}
    }
    ...
  }
```

Listing 4.7: Embedded Features in class `LegendGraphic` in `JFreeChart` project

Taking a look at the indicators, the following changes occurred:

Indicator 1 was totally redefined because the condition that the selector type must be boolean became part of the initial filter. Now, we look for clues, which indicate that an attribute might represent an optional feature, in the name of the selector: at least one selector has in its name strings suggesting an ON/OFF switch. That is it either contains one of the strings “flag”, “feature” or “switch” or it begins with “has” or ends with one of the “On” / “Off” strings.

Indicator 2 was left unchanged because, based on the experimental results, we concluded that it is the most relevant and well defined indicator of this design flaw.

Indicator 3 suggested that the suspected class should have a higher than average cyclomatic complexity. The idea of this indicator was good, but because the cyclomatic complexity metric is undefined in terms of a class it had to be rephrased: almost all affected methods of the class have a higher than average cyclomatic complexity; the average cyclomatic complexity should be project dependent.

In listing 4.7 the *draw* method of the *LegendGraphic* affected class is given as example. As can be seen, the class defines four boolean attributes. Although it is not visible in the listing, the attributes are written only in the constructor and in their setter methods and they are checked exclusively in simple *if* conditionals which are nested in order to combine the optional features of the class. This way the class fulfills the conditions mandated by the initial filter. In spite the fact that the first indicator is not present, both the second and third indicator were detected making this candidate a clear instance of the Embedded Features design flaw.

4.2 New Design Flaws

As pointed out in chapter 1.3, a first step in achieving the long term goal was done in [Tri08] which contains an appendix made up by ten design flaw definitions. This design flaw catalog is not even close to being complete. So, any new design flaw definition is more than well come to extend it.

The current work brings one such extensions to the design flaws catalogue and it is called “Duplicated Features”. The definition details of this design flaw can be can be seen by consulting section A.6 in the design flaws catalogue.

Chapter 5

CodeClinic4Eclipse

CodeClinic represents the practical results of [Tri08]. The functionality of the tool was already described in 2.2 and in this chapter we will present its implementation details.

5.1 The Framework

The tool was designed as a framework that can be extended using the standard Eclipse mechanism of extension points. Figure 5.1 depicts a very abstract logical view of the system architecture. The Eclipse plug-in that contains the actual application framework is shown on the left, and an extension plug-in that defines a single design flaw is shown on the right. The shaded component depicted below the two plug-ins belongs to the Java Developer Tools (JDT) of the Eclipse platform, and represents the structural model of the java source code project that is analyzed.

The main application is structured into three layers. The model layer holds the design flaw instances that are detected by the tool, together with information about the corresponding indicator presence and the current state of each instance. A design flaw instance can be in any of the three states: undecided (the initial state after detection), confirmed and rejected. The model persister is responsible for automatically persisting the entire model to disk upon closing the environment, and for restoring it back upon start-up. Thus, the tool supports the analysis of complex projects, across multiple sessions. The presentation layer depicted in the middle, is responsible for acting as an intermediary between the data model and the various possible views, as well as for accepting and responding to user gestures. For example, it is the responsibility of this layer to initiate the diagnosis process in response to the user's corresponding action in the graphical environment. The automated part of the diagnosis process is coordinated by the detection manager, with whom all concrete design flaw implementations are registered. The system supports an arbitrary number of design flaws, which can be packaged as separate eclipse plug-ins, and dynamically deployed on top of an existing installation.

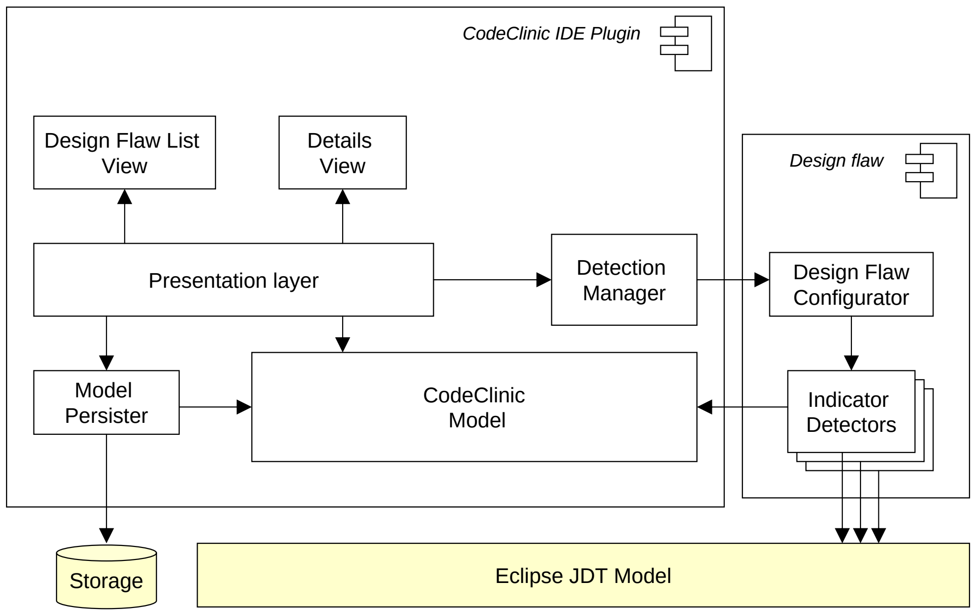


Figure 5.1: Architectural overview of CodeClinic

Figure 5.1 depicts the interactions that take place between the main application and one design flaw implementation. The design flaw configurator registers itself with the detection manager and configures the individual indicator detectors with appropriate parameters and metrics thresholds, which are configured by the user via the standard preferences dialog of the Eclipse platform. During analysis, individual indicator detectors receive access to the underlying structural model of the analyzed project, as well as to the CodeClinic model.

Finally, the top layer is represented by the views. The tool currently offers the two views mentioned above: the *design flaw* list view and the *details view*. In addition, for each candidate instances, a separate entry in the warnings list of the platform standard "Problems" view is added.

The architecture and the user interface of the tool are designed to allow an easy extension with further analysis features such as code visualizations, as well as the implementation of an advanced refactoring assistant. Using the refactoring capabilities of Eclipse, the refactoring assistant would guide the user through the reorganization process of each design flaw.

5.2 The Problem Model

In order to understand the way a design flaw is implemented, a short description of the problem model is needed. Figure 5.2 depicts a detailed view of the problem model. As

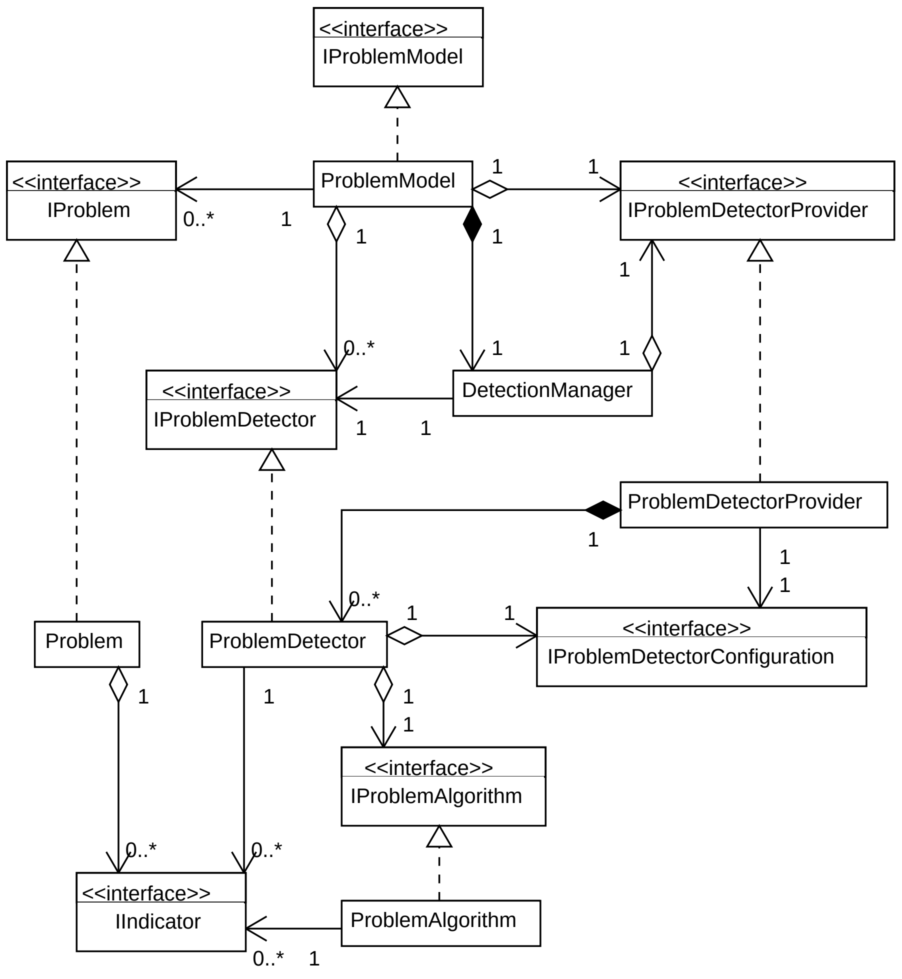


Figure 5.2: Detailed view of the problem modeling mechanism

pointed out in 5.1, the automated part of the diagnosis process is coordinated by the detection manager, with whom all concrete design flaw implementations are registered. The *IProblemModel* represents the interface with the presentation layer.

A design flaw is implemented in concordance with the specifications in the “Diagnosis Strategy” section of each design flaw definition. To add a new design flaw to the system, a new problem has to be added to the *configurations*[1] extension point. The problem needs a configuration class which has to extend *IProblemDetectorConfiguration*[2]. This class is responsible for creating the algorithm that will carry out the analysis needed for a design flaw.

The *createAlgorithm* method of each registered design flaw is called when the application is initialized and the created *IProblemAlgorithm*[3] instance is registered with the *DetectionManager*. The returned type has a method, *public IIndicator[] run(IJavaElement element) throws JavaModelException* which is called, for each registered algorithm, every time a detection of flaws is carried out. The list of indicators retuned by the method represents the list of detected indicators for the analyzed *IJavaElement*.

For now, there is a single implementation of the *IProblemAlgorithm* interface and that is the *ProblemAlgorithm* class. When creating an instance of this class, it has to be parameterized with a class that represents the structure which will passed to each indicator. The correlation between the elements described in the "Diagnosis Strategy" section and the *ProblemAlgorithm* is described bellow:

Search space: is set through the *setClassificationCheckers(IChecker... checkers)* method. The IChecker interface has a single method, *boolean check(IJavaElement element)* and is meant to check upon an element and return wheather it meets some specific condition or not.

Initial filter: is set through the *setInitialFilter* method. It takes only one argument which has to implement the *IConfigurableIndicatorEvaluator* [4] interface, a subtype of the *IIndicator*[5] interface.

Indicator: the indicators are added to the algorithm through the *addEvaluator* method. This takes exactly the same attribute type as the *setInitialFilterMethod*. The indicators are evaluated in the order they were added.

The support for *initial filter* represents an improvement brought by the current thesis to the application framework. When evaluating an element, it is considered a possible candidate only if it passes the initial filter. Even then, it must have at least one indicator present in order to be considered a real candidate.

In order for a "DesingFlawConfiguration" to create and configure a "ProblemAlgorithm", the following sequence of method calls will have to be invoked:

- Create a parameterized instance of ProblemAlgorithm:
  new ProblemAlgorithm<T>()

[1]de.fzi.codeclinic4eclipse.configurations
[2]de.fzi.codeclinic4eclipse.model.problem.IProblemConfiguration
[3]de.fzi.codeclinic4eclipse.model.framework
[4]de.fzi.codeclinic4eclipse.model.framework
[5]de.fzi.codeclinic4eclipse.presenter.model

- *setClassificationCheckers(IChecker... checkers)*
- *setStrategicParameterFactory(IStrategicParameterFactory<T>factory).* The *IStragtegicParameterFactory*[6] represents a factory that is parameterized with the same class as the problem algorithm and is used to create the structure representing the affected entity. As already pointed out, this structure is passed to the initial filter and each of the indicators when they are evaluated.
- Add the evaluators:
  addEvaluator(IConfigurableIndicatorEvalutor<? super T>evaluator).

[6]de.fzi.codeclinic4eclipse.model.framework

Chapter 6

Evaluation

This chapter is dedicated to evaluate the modifications and new features added to CodeClinic, through a series of case studies on intermediate to large sized software systems.

The evaluation process is similar to the one described in [Tri08]. This way, we have two experimental goals:

1. Check that the fully automated part works as intended. That is, verify if the pathological structure manifests itself through the candidates detected by CodeClinic and check if the design intent is characterized by a unique combination of indicators, and that the precision of the diagnosis process is directly proportional with the number of simultaneously detected indicators.
2. Evaluate the quality of each indicator by computing a linear regression model in order to estimate its relevance for its respective design flaw. The weights obtained from the linear regression constitute a predictive model that is useful in future analyses. The assumption that was made in [Tri08] is that the specified indicators, which represent the explanatory variables in the regression, are statistically independent.

The first experimental goal is needed because this way we can test if the detection of the modified design flaws has improved, while the second experimental goal will help us determine the quality of each indicator.

For the design flaws we have tried to improve, Collapsed Type Hierarchy and Explicit State Checks, our results will be compared against the results from [Tri08].

In the automated part of the diagnosis process only the detection of the pathological structure and design intent is covered while the questions intended to establish the strategic closure for the given fragment are not addressed. This is because the strategic closure can differ at various points in time due to the fact that it depends on momentary and planed requirements of the application. This is why the strategic closure will be ignored in the evaluation process.

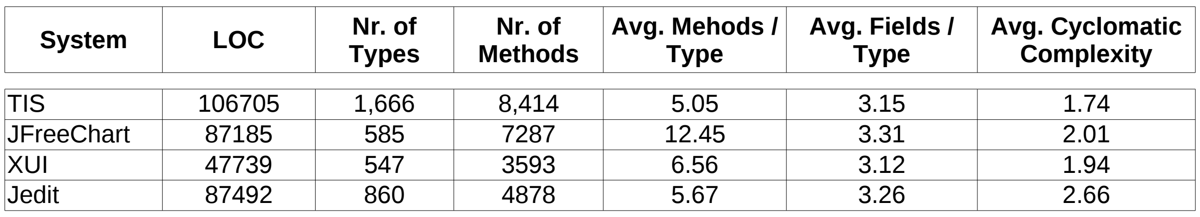
Therefore, an instance will be considered confirmed, if and only if the pathological structure and design intent match the specification of the flaw.

6.1 Experimental Setup

We try to achieve the experimental goals described above by testing CodeClinic on four intermediate to large software systems: three of them are open source systems and were used in the evaluation process of [Tri08]. The fourth system is an industrial system [1] produced by the Océ[2] corporation. Because of the corporation's non-disclosure policy, no information related to the analyzed product will be available.

A short description of each system used in the evaluation process is given bellow:

- *JFreeChart (v1.0.12):* [3] a free 100% Java chart library that makes it easy for developers to display professional quality charts in their applications. It supports many output types, including Swing components, image files (including PNG and JPEG), and vector graphics file formats (including PDF, EPS and SVG).
- *JEdit (v4.2):* [4] is a mature programmer's text editor with hundreds (counting the time developing plugins) of person-years of development. It has syntax highlighting and other specialized support for more than 130 languages, a built-in macro language and an extensible plugin architecture.
- *TIS:* a graphical application written in Java.
- *XUI (v1.0.4):* [5] a Java and XML based framework for building desktop, handheld, mobile, web and enterprise applications with a rich user interface. XUI provides development and debugging tools, as well as a set of look and feel components, widgets, and database bindings. In addition, the framework offers support for declarative XML based user interface generation.



System	LOC	Nr. of Types	Nr. of Methods	Avg. Mehods / Type	Avg. Fields / Type	Avg. Cyclomatic Complexity
TIS	106705	1,666	8,414	5.05	3.15	1.74
JFreeChart	87185	585	7287	12.45	3.31	2.01
XUI	47739	547	3593	6.56	3.12	1.94
Jedit	87492	860	4878	5.67	3.26	2.66

Figure 6.1: Size and complexity measurements for the analyzed systems

The table 6.1 shows some metrics we have calculated on the analysed systems using the tool *EclipsePro Audit*[6], version 6.0.0.

[1] The tested industrial system will be referred from now on as TIS (Tested Industrial System)
[2] http://www.oce.com
[3] http://www.jfree.org/jfreechart/
[4] http://www.jedit.org/
[5] http://www.xoetrope.com/xui
[6] http://www.instantiations.com/eclipsepro/audit.html

In the last column of the table is presented the average cyclomatic complexity metric of each system. This metric is used as a threshold value for the third indicator of the Embedded Features design flaw. In the indicator's definition is specified that the average cyclomatic complexity should be project dependent.

The other design flaws were tested with default threshold values.

6.2 Results

In this section we will discuss the statistical results collected during the evaluation process. In the first part we will discuss the results for the two design flaws that were intended to be improved by the current work and in the second part we will discuss the results for the newly implemented design flaws.

In the tables, like 6.2, used to achieve the first experimental goal, each of the individual rows, provides the statistics for those candidate flaws for which a particular number of indicators had been detected by our tool. This number is given in the first column of each table. The first table presents the results from [Tri08] while the second table presents the results of the current evaluation process. For example, in the second table, for the system JFreeChart, we have a total number of 15 candidate instances of the design flaw "collapsed type hierarchy". Based on the manual inspection of each candidate, only 5 candidates were confirmed, which means a precision of 33.3%. The last column provides the precision computed over all four analyzed systems.

Results of the Modified Design Flaws Figure 6.2 shows the results for the Collapsed Type Hierarchy design flaw.

As can be seen, we did not use in our evaluation process the InjectJ project. As a result we will only compare the results of the three systems used in both evaluation processes.

Comparing the results we can draw the following conclusions:

- For the case when only one indicator is present, the number of detected instances has decreased and, in the case of JFreeChart, the number of confirmed candidates has increased. Overall, the precision has increased.
- In the case when two indicators are present, although the overall precision has decreased, we can see that the number of detected instances has increased and the number of confirmed candidates is almost equal to the number of detected instances.

Figure 6.3 presents the results for the Explicit State Checks design flaw. It is obvious from the tables that overall the number of detected instances has decreased but we have lost some candidates that in [Tri08] were confirmed. The loss of candidates have happened because, previously, formal parameters and local variables were taken into consideration

Collapsed Type Hierarchy									
Nr. Ind.	XUI		JFreeChart		JEdit		InjectJ		Overall precision
	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	
1	1 / 3	33.3	3 / 19	15.8	2 / 6	33.3	1 / 5	20.0	21.2
2	0 / 0	-	0 / 0	-	2 / 2	100.0	0 / 0	-	100.0

Collapsed Type Hierarchy									
Nr. Ind.	XUI		JFreeChart		JEdit		TIS		Overall precision
	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	
1	0 / 1	0	5 / 15	33.3	1 / 5	20	3 / 6	50	34.6
2	1 / 1	100	4 / 4	100	4 / 5	80	0 / 0	-	90

Figure 6.2: Results for *Collapsed Type Hierarchy* design flaw

as state selectors in the “switch statements” code smell, which corresponds to this design flaw and because of the following bug that showed up in the SWITCH/IF detection mechanism: if the tool detects the first IF statement listed in 6.1, any other conditional constructs nested in the detected IF statement will be ignored.

```
if(...) {
    ...
    if(...) {
        ...
    } else {
        ...
    }
    ...
} else {
    ...
}
```

Listing 6.1: Example of undetected IF statement

Looking at the results and the conclusions we have drawn we can say that the detection of the Collapsed Type Hierarchy design flaw has obviously improved, while the Explicit State Checks design flaw will be improved only after the the above described bug will be resolved. This will be one of the first steps in our future work.

Regarding the experimental goals, we can see that the firs goal was achieved because the overall precision is directly proportional with the number of detected indicators.

Explicit State Checks									
Nr. Ind.	XUI		JFreeChart		JEdit		InjectJ		Overall precision
	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	
1	1 / 1	100.0	1 / 2	50.0	6 / 16	37.5	2 / 5	40.0	41.7
2	0 / 0	-	0 / 0	-	0 / 0	-	0 / 0	-	-

Explicit State Checks									
Nr. Ind.	XUI		JFreeChart		JEdit		TIS		Overall precision
	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	
1	0 / 0	-	2 / 7	28.6	0 / 3	0	1 / 1	100	27.3
2	0 / 0	-	0 / 0	-	0 / 0	-	0 / 0	-	-

Figure 6.3: Results for *Explicit State Checks* design flaw

In order to achieve the second experimental goal, we applied a simple linear regression on the set of observations made on the four analyzed systems. The obtained coefficients are given in figure 6.4, along with the corresponding values of the standard error, as computed by the statistics package R[7]. In both cases the second indicator seems to be the

Collapsed Type Hierarchy		
	I1	I2
Coeff.	0.486	0.364
Std. Err.	0.156	0.086

Explicit State Checks		
	I1	I2
Coeff.	1.000	0.200
Std. Err.	0.421	0.133

Figure 6.4: Computed coefficients for the modified design flaws

most reliable one, as indicated by the relatively low errors of the corresponding coefficients, while the first indicator remains quite unreliable. Though it would irrelevant to compare the current results with the ones determined in the evaluation process of [Tri08] because one of the analyzed projects is different, we can observer that using our results a coefficient could be computed for the first indicator of “explicit state checks” while using the results

[7]http://www.r-project.org/

from [Tri08] the statistics package R could not compute a coefficient for it. This is because in our case we had confirmed instances which presented the first indicator, but in [Tri08] no instances of “explicit state checks” with indicator 1 present were detected.

Results of the Newly Implemented Design Flaws Figure 6.5 displays the results for each new design flaw implemented in CodeClinic. Looking at the last column of each

Dispersed Control									
Nr. Ind.	XUI		JFreeChart		JEdit		TIS		Overall precision
	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	
1	0 / 0	-	1 / 1	100	1 / 1	100	0 / 0	-	100

Embedded Features									
Nr. Ind.	XUI		JFreeChart		JEdit		TIS		Overall precision
	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	
1	2 / 4	50	9 / 9	100	1 / 3	33.3	0 / 4	0	60
2	0 / 2	0	14 / 22	63.6	0 / 0	-	0 / 2	0	57
3	0 / 0	-	1 / 1	100	0 / 0	-	0 / 0	-	100

Embedded Strategy									
Nr. Ind.	XUI		JFreeChart		JEdit		TIS		Overall precision
	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	Conf. / Cand.	precision	
1	4 / 29	13.8	18 / 37	48.65	8 / 59	13.6	12 / 74	16.2	21.1
2	1 / 7	14.3	5 / 18	37.8	3 / 20	15	6 / 24	25	21.8
3	0 / 0	-	0 / 0	-	0 / 0	-	0 / 0	-	-

Figure 6.5: Overview of results for the newly implemented design flaws

table we can see that the first experimental goal was achieved for Dispersed Control and Embedded Features. This was not the case for Embedded Features. The precision for one detected indicator is 60%, for two indicators is 57% and for three indicators is 100%. The problem occurs when we have candidates with two indicator. This is a sign that one of the indicators has a weak definition or that it has little relevance. The coefficients and the corresponding values of the standard error for the three design flaws are given in figure 6.6. From these results, we can conclude that “dispersed control” defines the most reliable indicator because the error it’s coefficient has no error. For the reasons described above, the results for “embedded features” are less conclusive and should therefore be analysed and become subjects of enhancement. Finally, “embedded features” defines three indicator of which the third one is the most relevant while the first one could not be

Dispersed Control	
	I1
Coeff.	1.000
Std. Err.	0.000

Embedded Features			
	I1	I2	I3
Coeff.	0.325	0.571	0.103
Std. Err.	0.527	0.127	0.123

Embedded Strategy			
	I1	I2	I3
Coeff.	-	0.245	0.100
Std. Err.	-	0.046	0.029

Figure 6.6: Computed coefficients for the new design flaws

determined. This is because of the same reason described above: no confirmed instances which present the first indicator were detected.

Chapter 7

Conclusions

7.1 Summary of Contributions

By accomplishing the goals we have defined in 1, the research presented by the current work brings some contributions to the restructuring process, as reviewed bellow:

- **Improvements to the existing catalogue of design flaws**, by redefining some design flaws (*Collapsed Type Hierarchy*, *Embedded Strategy*, *Explicit State Checks*, *Dispersed Control* and *Embedded Features*) in order to make their detection better.
- **Extension of the existing catalogue of design flaws**, by defining a new design flaw with an associated restructuring pattern.
- **Implementation of new design flaws detection** into CodeClinic. This was done according to the specifications in the improved definitions.
- **Improvements of CodeClinic's framework**, by implementing the support for *initial filter* and by defining better structures that represent the elements affected by design flaws, which are passed to each indicator in order to perform different analysis.

7.2 Future Work

An immediate perspective, that we infer from the results obtained in the evaluation process, is that the detection mechanism of some design flaws needs further improvements. More specifically, we need to improve the detection of "SWITCH / IF" conditionals and the structures that are used to represent them as entities which are evaluated by the implemented indicators. The design flaws that are based on these structures are: "Collapsed Type Hierarchy", "Embedded Strategy", "Explicit State Checks" and "Embedded Features".

Because the design flaws in the existing catalogue represent implementation and restructuring guidelines, their definition should continuously be improved, until they are near to perfection.

The quality of a design flaws definition is directly proportional with the quality of its indicators. An issue that influences the quality of the indicators are is represented by the threshold values used to determine whether an entity has a specific symptom or not. They should be considered as targets of a future enhancement.

In CodeClinic there are eight design flaws implemented while in the existing catalogue we have eleven definitions. We aim to implement the other three design flaws in the future. This is important because the quality of a design flaw's definition can be determined only by evaluating its implementation.

Finally, further improvements and new features could be brought to CodeClinic, by integrating it with the refactoring engine provided by the development environment. The reorganization strategies, defined by each design flaw, could be implemented in the form of wizards to guide the developer through every step of the specified reorganization strategy.

Appendix A

Catalogue of Redefined Design Flaws

A.1 Collapsed Type Hierarchy

A.1.1 Description

According to [Rie96], the most natural way of expressing the specialization relationship (“is kind of”) between abstractions, is by implementing a type hierarchy using inheritance. In [Mey88], the author distinguishes between horizontal and vertical type generalization. Horizontal generalization is expressed through type parametrization, also known as generics. Specialization on the other hand, corresponds to vertical generalization, and is expressed through inheritance.

Thus, in an inheritance hierarchy, parent nodes represent vertically generalized abstractions of their children, which in turn are specializations of their parents. All members of the hierarchy support the interface of the root class, which can be used polymorphically.

A collapsed type hierarchy is the situation where an abstraction “absorbs” its own specializations, and emulates the specialization hierarchy, by explicitly checking the value assigned to a variable that represents the object’s special type. Figure A.1 depicts an example instance of a collapsed type hierarchy in a text editing system that handles ASCII and rich text documents. The class `Document` in the figure, provides the generic interface which defines common operations on generic documents (e.g. the methods `open()`, `copy()` and `paste()`). The implementation of the class makes use of a variable (e.g. `docType`) to track the current type of the document being processed. Its methods employ switch conditional constructs to inquire the current value of this variable in order to provide the needed specialized behavior.

A collapsed hierarchy makes understanding and changing individual specializations harder because their internal data and code are entangled in a single, bulky class. Also, it is hard to clearly distinguish general code from specialized code, and extend the hierarchy with new specializations.

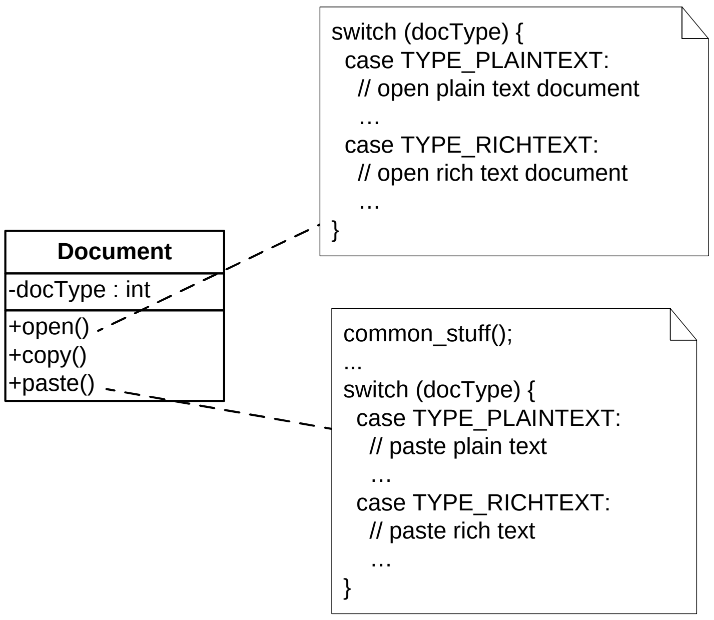


Figure A.1: An example of *collapsed type hierarchy*

A.1.2 Context

Design intent

You need to express a specialization hierarchy of a class, that represents a valid abstraction in the design of application. Clients of the root class need to access specialized versions in a transparent way, using the interface defined by the generic abstraction.

Strategic closure

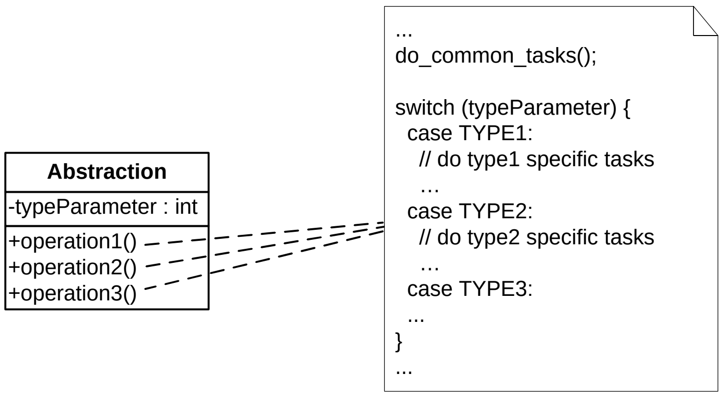
The number or implementation details of individual specializations is expected to change.

A.1.3 Imperatives

In order to maximize maintainability in the context described above, it is important to have a clean physical separation between the specializations themselves, as well as between what is common and what is characteristic for each specialization. This is most naturally achieved with the use of the inheritance relation. This will reduce the time needed to understand, add, change or remove individual specializations.

A.1.4 Pathological Structure

As exemplified in figure A.2, the entire hierarchy is collapsed into a single class which implements the root abstraction's interface. The implementations of the various operations



```
...
do_common_tasks();

switch (typeParameter) {
  case TYPE1:
    // do type1 specific tasks
    ...
  case TYPE2:
    // do type2 specific tasks
    ...
  case TYPE3:
  ...
}
...
```

Figure A.2: Pathological structure for *collapsed type hierarchy*

of this interface use a variable (either an attribute or a method parameter) for switching between the alternative behaviors. The attribute is usually initialized at the moment of instantiation and semantically embodies its concrete type. If method parameters are used, the clients pass arguments in order to request the expected behavior.

Although the concrete behavior of the abstraction instance is transparent to clients after its initialization, the design presented above has some serious drawbacks. The class that implements the abstraction is a monolith, in which specializations are tangled with one another in the implementation of every method defined by the generic interface. Because of this, the class increases in size and complexity. The entanglement on one hand and the increase in size and complexity on the other hand, make the design fragment hard to understand and change in the ways described in the context.

A.1.5 Reference Structure

The reference structure in the given context uses inheritance as the natural way to express a specialization hierarchy. As depicted in figure A.3, the type variable is no longer necessary, because the desired specialization is chosen through instantiation. Thus, the root abstraction received subclasses that correspond to each of the specializations. The large conditional constructs in the main abstraction have been dismantled branch by branch, and each branch has moved into the corresponding subclass. Each of the subclasses may implement any of the operations in the common interface in its own special way. Any

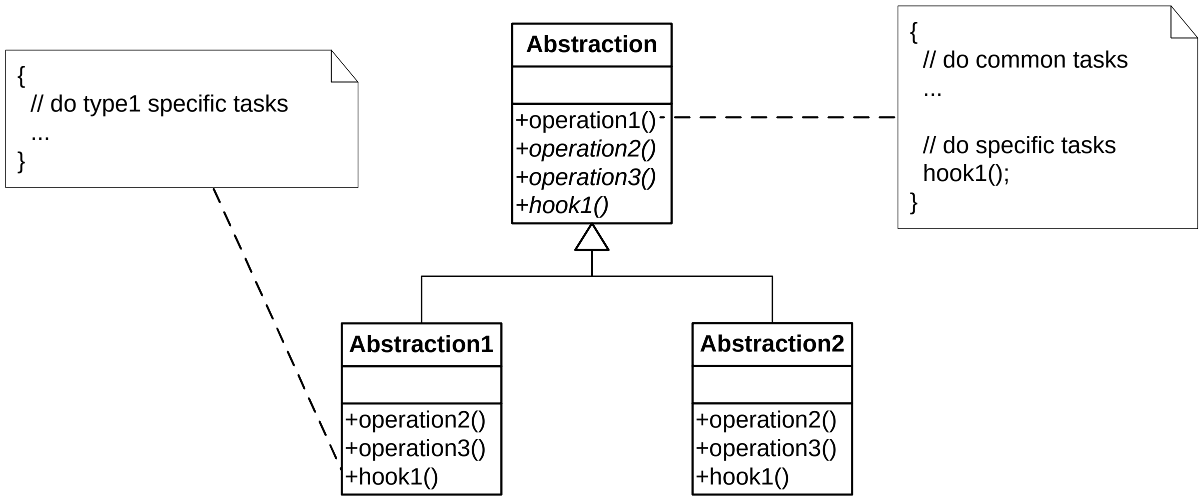


Figure A.3: Reference structure for *collapsed type hierarchy*

behavior that was common to all specializations has migrated into special hook methods, in concordance with the template method design pattern.

A.1.6 Diagnosis Strategy

Search space

All classes of the system.

Initial filter

A non trivial class that contains at least 1 switch or equivalent if-else constructs, located in separate methods, not using runtime type identification, on the same specific class attribute, which is not declared as final. The selector is compared against a set of *symbolic constants* (final and maybe static fields in the class) or *constants defined in an enum* (all the constants belong to the same enum type).
If the conditional construct is an *if* construct, then the checked *expression* should be **selector == symbolic_constant** or **getSelector() == symbolic_constant**.

Indicators

Indicator 1: *The name of the parameter used in the conditional expressions contains the word "type", thus suggesting a type variable. Alternatively, the parameter is compared against a set of symbolic constants whose names contain such a word.*

Indicator 2: *Usage patterns of the parameter used in the conditionals suggest a type variable. In other words, there is no write access on the variable, anywhere in the class, with the exception of object constructors and the parameter's setter.*

Context matching

Question 1: *It must be confirmed that the class represents a valid abstraction in the design of the system.*

Question 2: *It must be confirmed that the parameter used in the conditional constructs represents a type code, used for implementing a specialization hierarchy of the abstraction represented by the class.*

Question 3: *It must be confirmed that the behavior being specialized semantically describes the abstraction in its entirety. In other words, the behavior that is being specialized corresponds to the abstraction modeled by the class in its entirety, not to a limited aspect of its implementation.*

Question 4: *It must be confirmed that the number or implementation details of individual specializations is expected to change.*

A.1.7 Reorganization Strategy

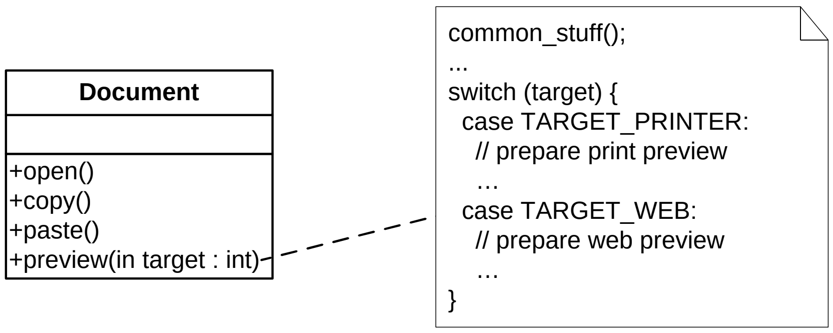
```
1: Based on the range of allowed values of the parameter, identify specializations of the
   class
2: if class has no subtypes then
3:     Apply refactoring “replace type code with subclasses” [Fow99] for the identified
       specializations
4:     Apply refactoring “replace conditional with polymorphism” [Fow99] for the newly
       created subclasses
5: else
6:     Apply design pattern “bridge” [GHJV96] to extract the collapsed hierarchy into a
       parallel inheritance hierarchy
7: end if
8: Push up common behavior as high as possible in the newly created inheritance hierar-
   chy, by creating template methods, in accordance with the design pattern “template
   method” [GHJV96]
```

A.2 Embedded Strategy

A.2.1 Description

In object oriented design, functionality is distributed among objects, based on the identity and perceived responsibilities of their corresponding classes. Thus, objects can be seen as actors that cooperate in order to realize the system's functions. Oftentimes, one class instance needs to be able to vary some detail of its behavior dynamically, based on the momentary needs of its clients. Normally, this can be achieved elegantly by employing the strategy design pattern. It allows defining a family of interchangeable algorithms in the form of a hierarchy. A client uses one member of this hierarchy in order to dynamically configure the class instance. Thus, the configurable instance is said to provide a context for the algorithm used as a parameter.

An embedded strategy is the situation in which the class providing the context, explicitly switches between alternative algorithms, whose implementations are all hard-coded into the class itself. Figure A.4 depicts an example instance of an embedded strategy, in a hypothetical text editing application.



```
common_stuff();
...
switch (target) {
  case TARGET_PRINTER:
    // prepare print preview
    …
  case TARGET_WEB:
    // prepare web preview
    …
}
```

Figure A.4: An example of *embedded strategy*

The class *Document* in the figure, provides the generic interface which defines common operations on generic documents, such as *open()*, *copy()*, *paste()* and *preview()*. Clients configure the preview operation by choosing between two alternative types of algorithms: one for print preview, the other for web preview. The *preview()* method uses a conditional construct in order to select the desired algorithm at runtime.

An embedded strategy makes understanding and changing both context class and individual algorithms harder, because their internal data and code are entangled in a single, bulky class. Also, it is hard to clearly distinguish general code from algorithm-specific code, as well as to implement additional algorithms.

A.2.2 Context

Design intent

Allow clients of a class that represents a valid abstraction in the application's design, to dynamically configure its instances with a family of interchangeable algorithms, that contribute to part of the services provided by the class.

Strategic closure

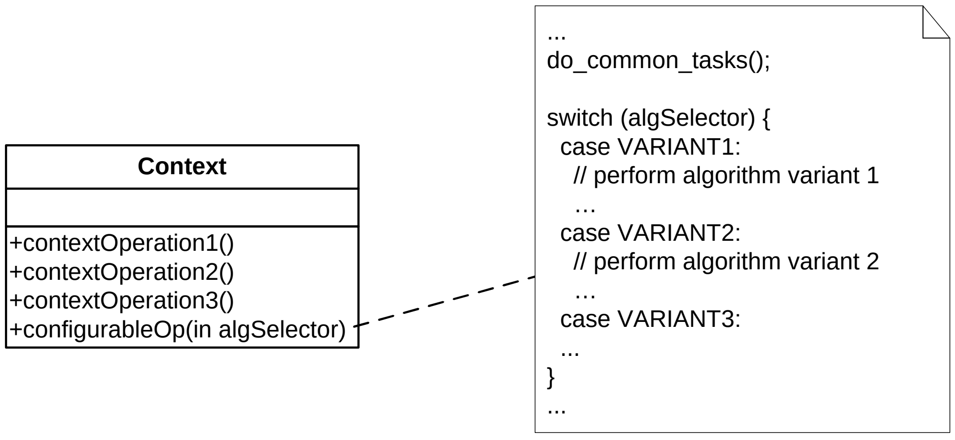
You expect the class providing the context, or the number and implementation of individual algorithms in the family to change independently.

A.2.3 Imperatives

From a maintainability standpoint, if the context and the algorithms are expected to change independently, it is important to have a clean separation between their implementations. Furthermore, in order to ease understanding and changing individual algorithms in isolation, it is important to cleanly separate their implementations from one another while simultaneously avoiding code duplication. Since the alternative algorithms semantically represent specialized versions of an abstract generic algorithm, we have a specialization hierarchy, that is best expressed using inheritance.

A.2.4 Pathological Structure

The pathological structure is depicted in figure A.5. All alternative algorithm implemen-



```
...
do_common_tasks();

switch (algSelector) {
  case VARIANT1:
    // perform algorithm variant 1
    ...
  case VARIANT2:
    // perform algorithm variant 2
    ...
  case VARIANT3:
    ...
}
...
```

Figure A.5: Pathological structure for *embedded strategy*

tations are contained within and mixed with the implementation of the context class. Clients choose between the various algorithms by means of selectors, defined either as class attributes or as parameters of those methods that support configuration. In order to select the desired behavior, the methods themselves use conditional constructs that explicitly check the value of the selector.

Although the class allows its clients to dynamically configure part of an object's behavior, the design presented in figure A.5 clearly disregards the imperatives described above. The implementation details of the algorithms are entangled with those of the context class, which impedes understanding and changing any of the two in isolation. In addition, the danger of having duplication between methods that rely on the same algorithms is very high.

A.2.5 Reference Structure

In the reference structure for the described context, the algorithm family is extracted out of the class providing the context, and modeled by a specialization hierarchy. This corresponds to the "strategy" design pattern. As depicted in figure A.6, the selector pa-

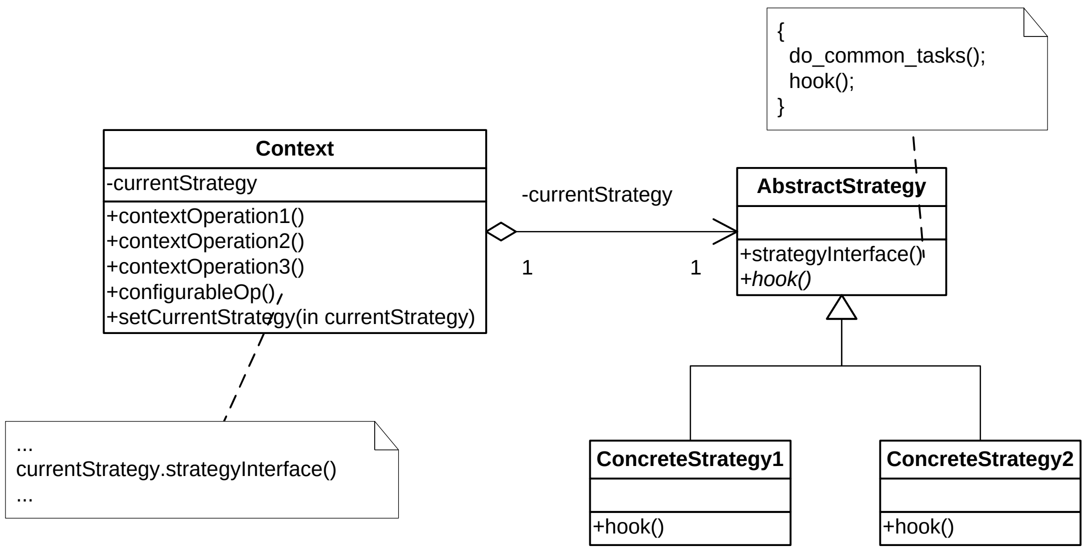


Figure A.6: Reference structure for *embedded strategy*

rameter is no longer needed, because explicit checks are now replaced by polymorphic calls to an abstract strategy interface. The individual branches of the former conditionals have migrated into the corresponding member of the newly defined specialization hierarchy. Clients can either use a dedicated method for configuring the context, as shown in the figure, or can pass a reference to the desired strategy directly, upon each call. The use of inheritance allows extracting commonalities into the upper layers of the hierarchy, by employing the template method pattern.

A.2.6 Diagnosis Strategy

Search space

All classes in the system.

Initial filter

A class contains at one or more methods that employ switch or equivalent if-else constructs with at least two branches, not using runtime type identification, which check a particular formal parameters having the same name. The value of the selector is not modified in the method. The checked method parameters will be referred to as selectors.

Indicators

Indicator 1: *The name of the selector contains the word "strategy" or "algorithm", thus suggesting a strategy configuration parameter. Alternatively, the selector is compared against a set of symbolic constants whose names contain such a word.*

Indicator 2: *The body of each branch represents an algorithm. Thus, method calls should be to private methods of the class. That is, all method calls should be to private methods of the containing class, except the getters and setters.*

Indicator 3: *The concern that is being specialized represents a small fraction of the class. In other words, a very small number of the class' non-accessor methods contain the affected conditional constructs. If the conditional construct is contained in more that one method, there should be signs of code duplication between the bodies of the same branches.*

Context matching

Question 1: *It must be confirmed that the classes represents a valid abstraction in the design of the system.*

Question 2: *It must be confirmed that the parameter used in the conditional constructs represents a type code, used for implementing a specialization hierarchy.*

Question 3: *It must be confirmed that the abstraction being specialized does not correspond to the concept modeled by the class as a whole, but to some limited aspect of its implementation. The rest of the class can be regarded as the context in which this family of related algorithms perform some limited task.*

Question 4: *It must be confirmed that changes to class providing the context as* *well as the number and implementation of individual algorithms in the embedded hierarchy are likely to happen in isolation.*

A.2.7 Reorganization Strategy

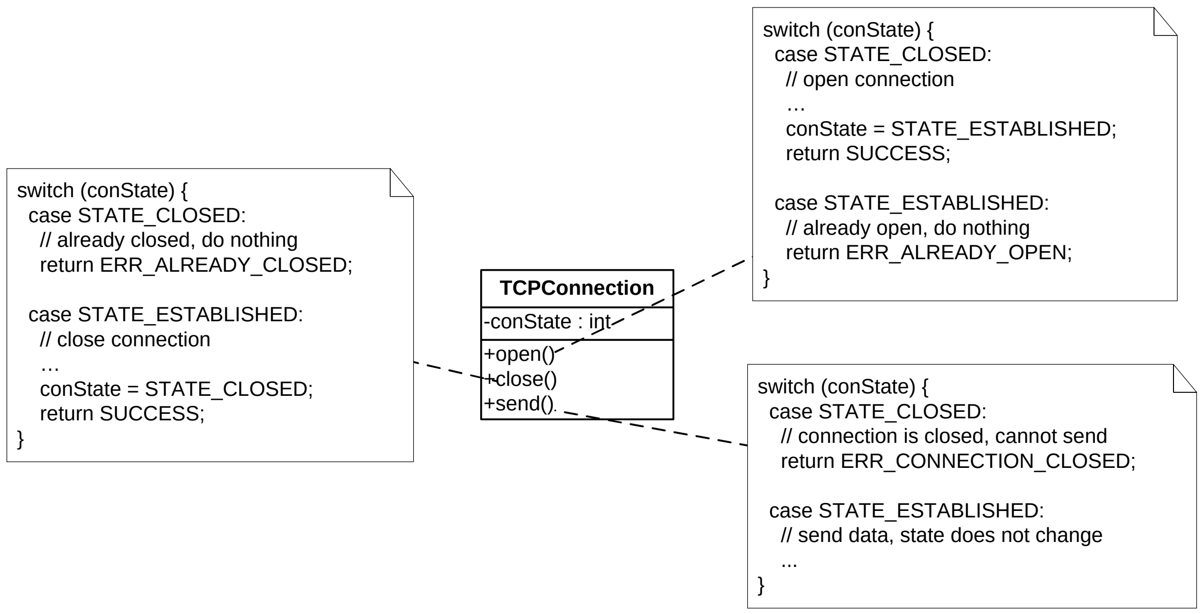
```
1: Identify all abstract strategy types based on the logic of the conditional structures
2: for all strategy interfaces $IS_i$ do
3:     Based on the range of values taken by the type parameter, identify all concrete
       strategies that correspond to $IS_i$
4:     Apply refactoring "replace conditional logic with strategy" [? ] to implement the
       "strategy" design pattern [GHJV96] corresponding to $IS_i$
5:     Push up common behavior as high as possible in the newly created strategy hierar-
       chy, by creating template methods, in accordance with the design pattern "template
       method" [GHJV96]
6: end for
```

A.3 Explicit State Checks

A.3.1 Description

In object oriented programming, polymorphism is the universal mechanism that allows an abstraction, defined by its interface, to vary behavior transparently with respect to its clients. In particular, polymorphism is the most natural way of altering an object's behavior, as observed by its clients, based on its current "state". Normally, under the notion of "state", we understand a snapshot of the current values of all attributes defined in the class. In this case however, "state" must be understood in an abstract sense, as the state of a domain abstraction that is modeled by the class.

The design flaw "explicit state checks" refers to the situation in which an object uses explicit checks on some internal piece of data, in order to execute state specific behavior or manage its "state" transitions. The data that is checked represents the current "state" at any given time.



```
switch (conState) {
  case STATE_CLOSED:
    // already closed, do nothing
    return ERR_ALREADY_CLOSED;

  case STATE_ESTABLISHED:
    // close connection
    ...
    conState = STATE_CLOSED;
    return SUCCESS;
}
```

```
switch (conState) {
  case STATE_CLOSED:
    // open connection
    ...
    conState = STATE_ESTABLISHED;
    return SUCCESS;

  case STATE_ESTABLISHED:
    // already open, do nothing
    return ERR_ALREADY_OPEN;
}
```

```
switch (conState) {
  case STATE_CLOSED:
    // connection is closed, cannot send
    return ERR_CONNECTION_CLOSED;

  case STATE_ESTABLISHED:
    // send data, state does not change
    ...
}
```

Figure A.7: An example of *explicit state checks*

In the example depicted in figure A.7, the class `TCPConnection` uses the attribute `conState` to store the current connection state. The class implements a number of operations (e.g. `open()`, `close()`, `send()`) whose behavior varies according to this state. The selection occurs explicitly, by checking the attributes current value upon each call. Some operations change the current state of the object by assigning a new value to the attribute.

Explicit state checks make it harder for the maintainer to add new states, as well as identify and change behavior that is specific to a given state. Furthermore, it is hard to distinguish state dependent from state independent code in the bloated classes that suffer from this design flaw.

A.3.2 Context

Design intent

Objects of a class that represents a valid abstraction in the application's design, need to vary their behavior dynamically, based on an abstract state, that can be managed either internally or externally.

Strategic closure

You expect the number of states to change, changes to the code that corresponds to individual states, or changes that would require the maintainer to distinguish state dependent from state independent behavior.

A.3.3 Imperatives

In order to maximize maintainability in the context described above, we must isolate on one hand, state dependent from state independent code from one another, and on the other hand, code that is specific for each individual state from one another. In addition, we can minimize the risk of code duplication by extracting commonalities in behavior among various states, in a specialization hierarchy.

A.3.4 Pathological Structure

Figure A.8 illustrates the most important characteristics of the pathological structure in the case of "explicit state checks". As shown in the figure, both state dependent and state independent code are contained inside a single monolithic class. The current state of an instance is held by an attribute that usually has some enumerated type. Throughout the implementation of the class, the value of this attribute is repeatedly checked inside typically large conditional constructs in order to select the desired behavior. State changes are carried out by assigning a new value to the attribute.

The pathological design clearly contradicts the imperatives described above. The entire functionality is entangled inside a single class, with several bloated methods. This leads to increased effort and error proneness in understanding and changing both state dependent and state independent code.

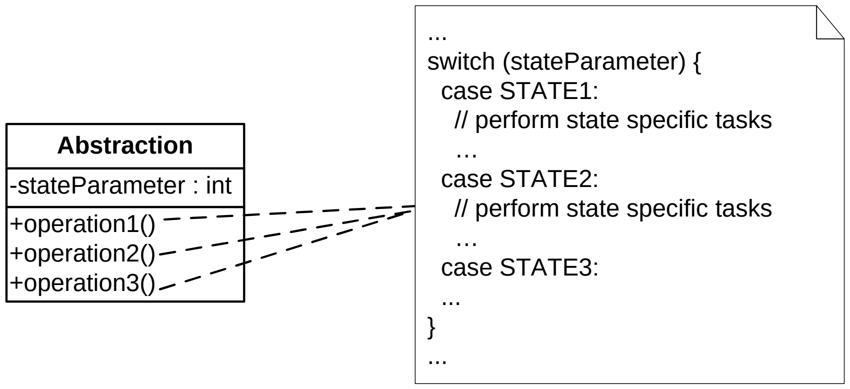


Figure A.8: Pathological structure for *explicit state checks*

A.3.5 Reference Structure

In the given context, the reference structure corresponds to the design pattern “state”, as illustrated in figure A.9. In the structure presented in the figure, state specific be-

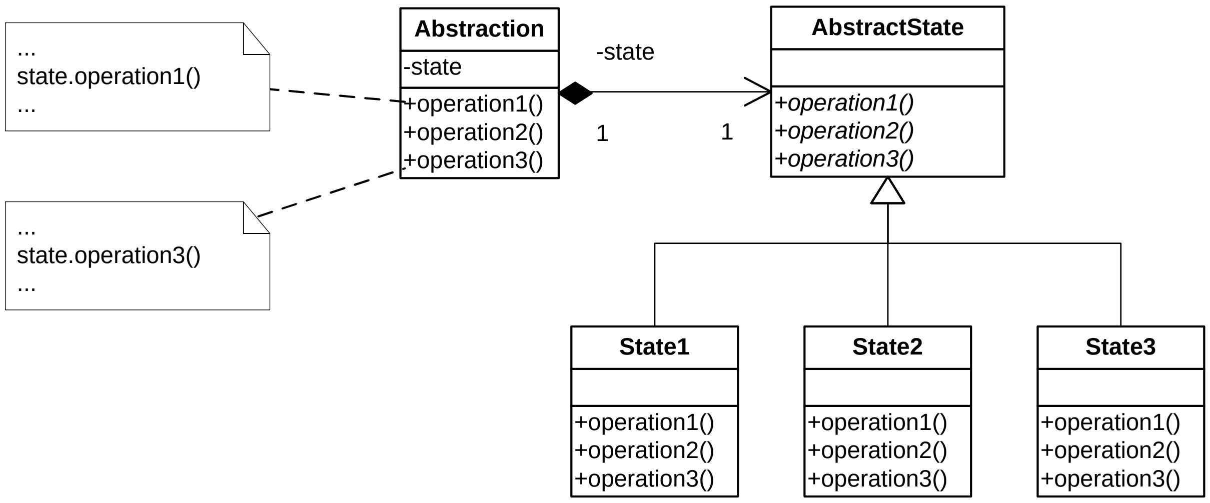


Figure A.9: Reference structure for *explicit state checks*

havior has been isolated into separate classes, that form a specialization hierarchy. As a result, the large conditional constructs in the main class had been dismantled branch by branch, and each branch has moved into one of the concrete state classes. The main class aggregates the root of the specialization hierarchy, defining the common state interface.

State management including instantiation of the state objects is generally best performed internally by the main class.

A.3.6 Diagnosis Strategy

Search space

All classes of the system.

Initial filter

A non trivial class that contains at least two switch or equivalent if-else constructs, located in separate methods, not using runtime type identification, on the same specific class attribute, which is not declared as final. The selector is compared against a set of *symbolic constants* (final and maybe static fields in the class) or *constants defined in an enum* (all the constants belong to the same enum type). If the conditional construct is an *if* construct, then the checked *expression* should be: **selector == symbolic_constant** or a composite expression, like **class_attribute_1 == symbolic_constant ¡boolean_operand¿ . . . class_attribute_n == symbolic_constant**.

Indicators

Indicator 1: *The name of the checked attribute contains the word "state", thus suggesting a state variable. Alternatively, the switch parameter is compared against a set of symbolic constants whose names contain such a word.*

Indicator 2: *Usage patterns of the switch parameter suggest a state variable, in the sense that the value of the checked parameter is changed, either within branches of the conditional constructs, or from the clients that called the respective operation, either direct or through setter*

Context matching

Question 1: *It must be confirmed that the classes represents a valid abstraction in the design of the system.*

Question 2: *It must be confirmed that the parameter used in the conditional constructs semantically denotes the state of the domain abstraction.*

Question 3: *It must be confirmed that the number or implementations of individual state specific behaviors are expected to change, or changes are expected that would require the maintainer to distinguish state dependent from state independent code.*

A.3.7 Reorganization Strategy

```
1: if state management occurs from within the conditional constructs described in indicator 1 then
2:     Apply refactoring “replace state-altering conditionals with state” [? ] on the affected class
3: else
4:     Based on the range of allowed values of the state parameter, identify the range of possible states
5:     Apply refactoring ”replace type code with state” [Fow99]
6:     Implement a state management interface in the context class and adapt clients to use the context’s state management interface
7:     If desired, optimize state management performance, by replacing on demand state object instantiation with pre-instantiated state objects, according to the “singleton” design pattern [GHJV96]
8: end if
9: Push up common behavior as high as possible in the newly created state hierarchy, by creating template methods, in accordance with the design pattern “template method” [GHJV96]
```

A.4 Dispersed Control

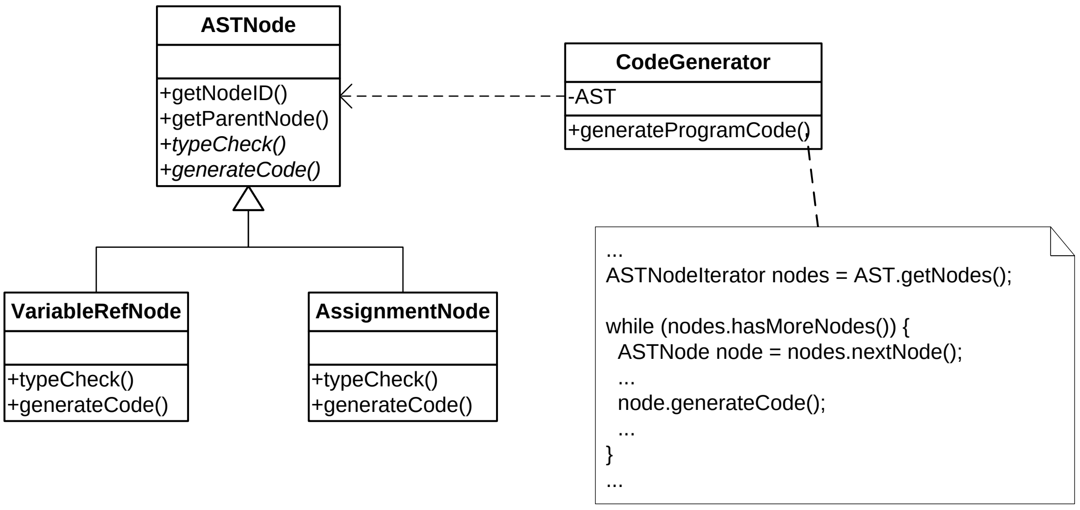
A.4.1 Description

One of the fundamental differences between the procedural and the object oriented paradigms, is the way in which complex operations are broken up into pieces that are allocated to various program functions. In the procedural world, we have a workflow, or activity based view, according to which a complex operation is broken up based on the logical steps in the workflow. In the object oriented world, a decomposition based on object identity and responsibilities is dominant while the activity based decomposition still plays an important role within classes.

There are however situations which justify concentrating bits of functionality from heterogeneous classes, giving priority to an activity rather than identity based decomposition. The design pattern "visitor" corresponds to such a situation, where the realization of functionalities on top of complex structures of related objects presumes a certain degree of orchestration between type specific behaviors. In order to make the implementation of complex operations that semantically pertain to an entire structure of objects maintainable, the individual bits of type specific behavior are encapsulated into a so called visitor class, that represents the high-level operation.

We have a case of "dispersed control", when in a situation such as the one described above, the implementations of structure related functionalities are broken up and dispersed between the individual types that belong to the structure. Figure A.10 illustrates an example of dispersed control in a hypothetical compiler. The heterogeneous structure is represented in this case by the abstract syntax tree, which is modeled as a collection of specialized `ASTNode` objects. Code generation represents the high level operation that is defined for the structure. Its implementation is dispersed throughout the `generateCode()` method in the entire `ASTNode` hierarchy. An external client to the hierarchy, called `CodeGenerator`, orchestrates the code generation functionality by iterating through the objects in the structure in a given way, and calling each object's individual version of `generateCode()`.

Since the types of individual nodes in the abstract syntax tree depend on the programming language that is being compiled, it is reasonable to expect that once implemented, the hierarchy will not change significantly. On the other hand, it is very probable that further global operations on the abstract syntax tree, such as type checking for example, will have to be added or changed frequently in the future. But maintaining such operations as well as adding new ones in this structure is difficult, because all descendants of `ASTNode` need to be adapted every time. In addition, individual types in the `ASTNode` hierarchy are harder to understand and changed, because their implementation is cluttered with the various bits of functionality that realize each global operation.



```
...
ASTNodeIterator nodes = AST.getNodes();

while (nodes.hasMoreNodes()) {
  ASTNode node = nodes.nextNode();
  ...
  node.generateCode();
  ...
}
...
```

Figure A.10: An example of *dispersed control*

A.4.2 Context

Design intent

You need to implement a number of operations that semantically pertain to a complex, heterogeneous structure, which consists of objects of classes, that represent a valid specialization hierarchy. The operations accumulate data from the structure elements, or perform some global function, by orchestrating between element specific bits of functionality.

Strategic closure

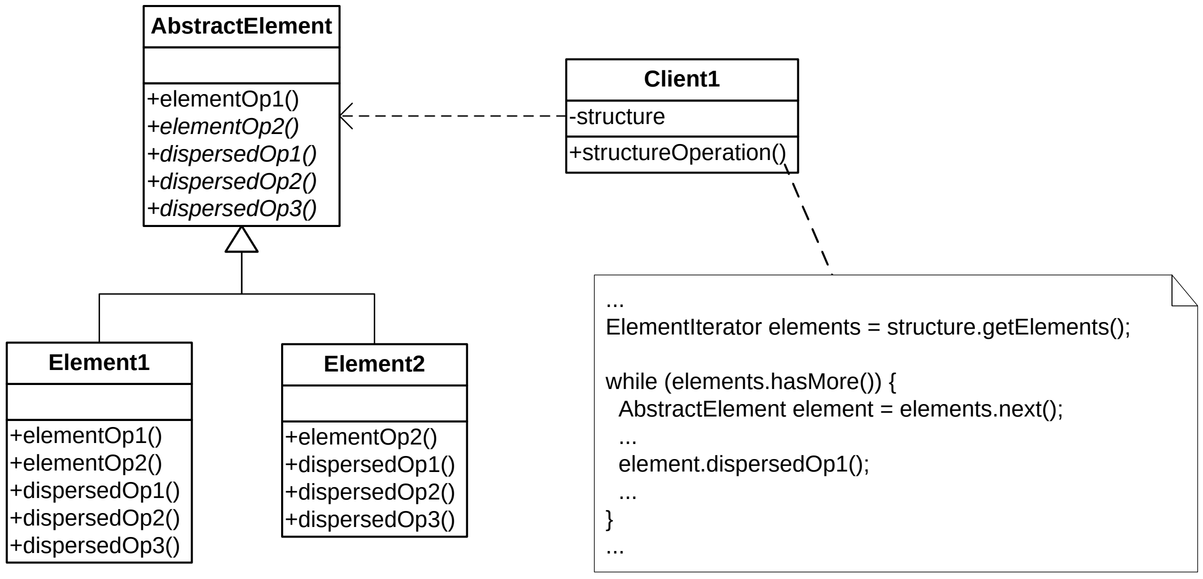
The number and implementation of the global operations is expected to change more frequently than the number and implementation of the individual element types forming the structure.

A.4.3 Imperatives

In order to maximize maintainability in the context described above, the implementation parts that are expected to change often (i.e. the global operations on the structure) must be decoupled from one another, and from those that are expected to change more rarely (i.e. the elements of the structure). In other words, the activity based view of complex operations should be given priority over the identity based view.

A.4.4 Pathological Structure

As shown in figure A.11, the inheritance hierarchy that describes the elements of the object structure, is also used for expressing the differences that exist between type specific bits of the global operation. In other words, every element's type interface is a mix



```
...
ElementIterator elements = structure.getElements();

while (elements.hasMore()) {
  AbstractElement element = elements.next();
  ...
  element.dispersedOp1();
  ...
}
...
```

Figure A.11: Pathological structure for *dispersed control*

of both operations that are "local" to each element type, and operations that perform element specific bits of a greater, global operation that relates to the entire structure. The orchestration that is necessary in order to obtain the end result from the combination of individual element specific bits, is realized in an external client, which iterates over the structure in an appropriate way. Oftentimes, the purpose of the global operation is to derive some information that results from accumulating information from each structure element.

The pathological structure described above disregards the imperatives of this design flaw, because it mixes global, structure specific and element specific functionality, cluttering the implementation of all element types. Furthermore, individual structure related operations are hard to understand and change, because their implementation is dispersed throughout the entire element hierarchy. Adding a new global operation is hard, because it requires adding new methods to all element types.

A.4.5 Reference Structure

The reference structure is represented by the design pattern "visitor" and is illustrated in figure A.12. All individual bits defining a complex operation that pertains to the whole

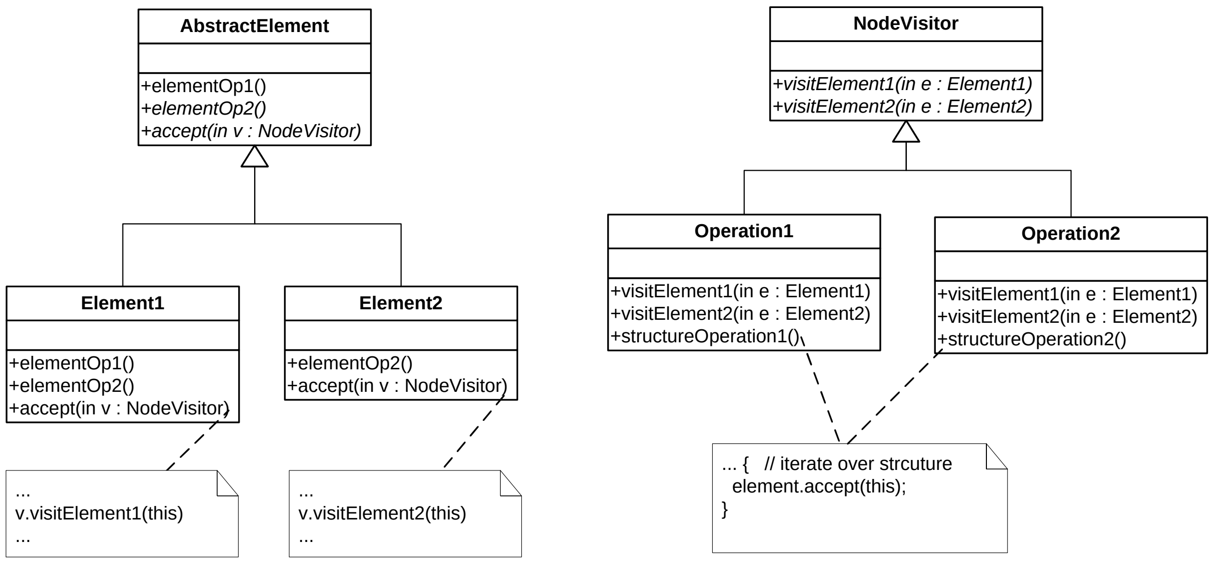

AbstractElement
+elementOp1()
+elementOp2()
+accept(in v : NodeVisitor)
Element1
+elementOp1()
+elementOp2()
+accept(in v : NodeVisitor)
Element2
+elementOp2()
+accept(in v : NodeVisitor)
...
v.visitElement1(this)
...
...
v.visitElement2(this)
...
NodeVisitor
+visitElement1(in e : Element1)
+visitElement2(in e : Element2)
Operation1
+visitElement1(in e : Element1)
+visitElement2(in e : Element2)
+structureOperation1()
Operation2
+visitElement1(in e : Element1)
+visitElement2(in e : Element2)
+structureOperation2()
... { // iterate over strcuture
element.accept(this);
}

Figure A.12: Reference structure for *dispersed control*

structure, have been encapsulated in separate visitor classes, which as special cases of visitors, are all part of a common specialization hierarchy. Each visitor class corresponds to exactly one global operation. Furthermore, a double dispatch mechanism has been implemented between the two hierarchies, which gives the maintainer complete flexibility in changing the number and implementations of individual visitors, without touching the element hierarchy. The element hierarchy defines a generic interface that allows a client to use an arbitrary visitor object, and each visitor object defines element specific methods, that are called during visitation. In addition, all element types need to provide an interface that allows a visitor to access element internal data. This break of element encapsulation is the price paid for the increased maintainability of the global operations.

A.4.6 Diagnosis Strategy

Search space

All type hierarchies in the system.

Initial filter

The hierarchy should have at least one client. The root of the hierarchy defines a number of methods, either concrete or abstract, that are either overridden or implemented in almost all terminal nodes of the hierarchy. The overridden methods represent bits of semantically unrelated global operations. Thus, there are no calls between any pair of such methods.

Indicators

Indicator 1: *The overridden methods represent bits of semantically unrelated global operations. Thus, clients do not call more than exactly one such method, from within any of their methods. But different methods of the hierarchy can be called from within the same client's different methods.*

The overridden methods are called from within contexts that suggest an orchestration, or an accumulation of information. An orchestration is probable if calls to many such methods occur from within a cycle, in which the target object of the call is continually changed. Accumulation is probable if many such methods return information by means of a return value or output parameter.

Context matching

Question 1: *It must be confirmed that the hierarchy represents a valid specialization hierarchy in the application's design.*

Question 2: *It must be confirmed that the identified group of methods represent bits of global operations that are semantically associated to a complex structure, formed with instances of the hierarchy. The clients that call methods in the identified group, orchestrate the calls to individual objects in the structure in order to accumulate information or otherwise perform a structure specific service.*

Question 3: *It must be confirmed that the number and implementation of the global operations is expected to change more frequently than the number and implementation of the individual element types forming the structure.*

A.4.7 Reorganization Strategy

```
1: if there is no double dispatch infrastructure in place then
2:     Create abstract visitor class $AV$
3:     for all types $T_i$ that form the heterogeneous structure do
4:         Create a corresponding visit<...> method in $AV$
5:         Create a method named accept(...) that receives a reference of $AV$'s type as
           a parameter, and calls the proper visit<...> method on the received reference
6:     end for
7: else
8:     Let $AV$ be the root of the existing visitor hierarchy
```

```
 9: end if
10: for all dispersed operations $O_i$ in the heterogeneous hierarchy do
11:     Create a concrete visitor class $V_i$, as a subtype of $AV$
12:     Apply refactoring "extract method" [Fow99] on the orchestration code within the
        client class, containing the call to $O_i$
13:     Move previously extracted method to $V_i$
14:     Replace the call to $O_i$ with a call to the corresponding accept(...) method,
        passing the reference this as argument
15:     Move all element specific method implementations that override or implement $O_i$,
        to the various visit<...> methods in $V_i$. If needed provide accessor methods to
        internal attributes of the heterogeneous element types
16:     Remove all empty methods corresponding to $O_i$ from the element hierarchy
17: end for
```

A.5 Embedded Features

A.5.1 Description

Inheritance allows the definition of a hierarchy of specialized versions of an abstraction. By using a reference to the root type, other classes in the system can use any of the specialized variants without knowing their exact type. Furthermore, the object that the reference points to, may be dynamically exchanged with other objects of any of the specialized types.

But what if we wanted to dynamically give or withdraw new responsibilities or features, to one particular object? We cannot achieve this using inheritance alone, because object state would be lost every time we replaced instances. Furthermore, if the number of individual features were large, and we wanted to combine them, the number of specializations, and therefore classes in the system, would explode uncontrollably. The most flexible solution for this scenario involves combining both the inheritance and composition mechanisms, in accordance with the decorator design pattern.

A class is said to suffer from "embedded features", if it uses attributes that represent on/off switches for optional features of the class. The attributes are explicitly checked in order to choose the desired behavior in each case. Figure A.13 illustrates this situation.

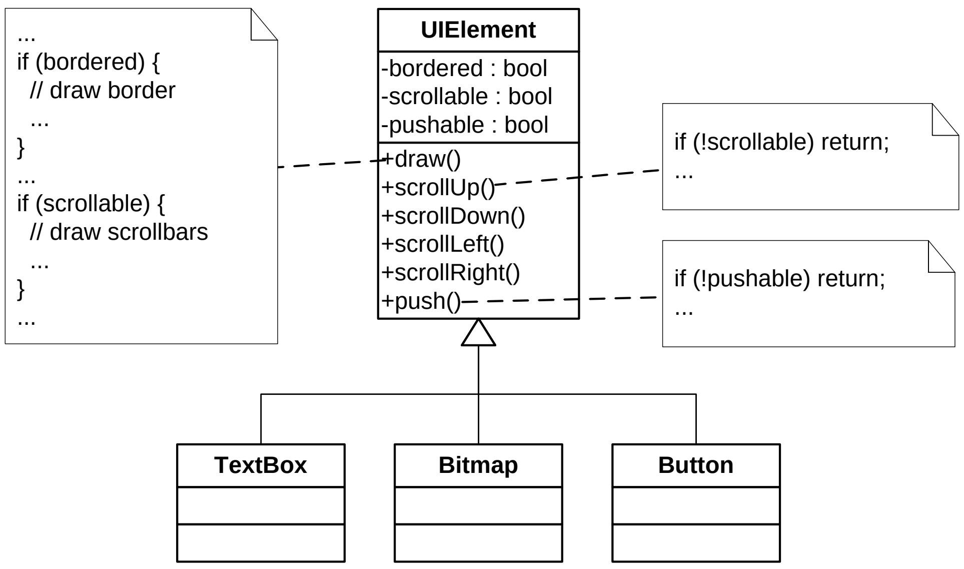


Figure A.13: An example of *embedded features*

In the example shown in the figure, the abstractions that need to be configurable are elements of a graphical user interface, whose common root class is `UIElement`. The features that an `UIElement` is expected to support are: drawing a 3D border around the element (the attribute `bordered`), scrolling support (the attribute `scrollable`), and support for push actions (the attribute `pushable`). As we can see in the figure, the implementations of the element's operations contain blocks that check the current status of each feature in order to provide the desired behavior.

The structure described above negatively affects maintainability, because it mixes code that belongs to the features with the code of the base abstraction. Thus, it becomes harder and harder to add new features, as well as understand and change any of the features in isolation. In addition, in the particular situation depicted in figure A.13, it is not possible to change the "layering" of the bordering and scrolling features, without changing the implementation. Thus, we must for example statically decide between having the scroll bars outside, or inside of the 3D border.

A.5.2 Context

Design intent

You want to allow clients to dynamically enable or disable one or more optional features on instances of a class, or family of classes. The class, or hierarchy of classes represent a valid abstraction or specialization hierarchy in the application's design.

Strategic closure

You expect further features to be added in the future, changes to occur to the existing features, or you need to be able to dynamically change the layering of the features. The public interface of the base abstraction is either not expected to suffer frequent changes, or changes are only expected in methods that are not related to any of the optional features.

A.5.3 Imperatives

In order to maximize maintainability in the context described above, we need to separate the implementation of the base abstraction from the implementations of the features, and the implementations of the features from one another. In order to maximize the flexibility in combining several features, the choice of layering should be left entirely to the clients, and not hardwired in the base abstraction.

A.5.4 Pathological Structure

The pathological structure of the flaw is very simple, and is schematically depicted in figure A.14. The operations of the base abstraction employ simple conditionals which

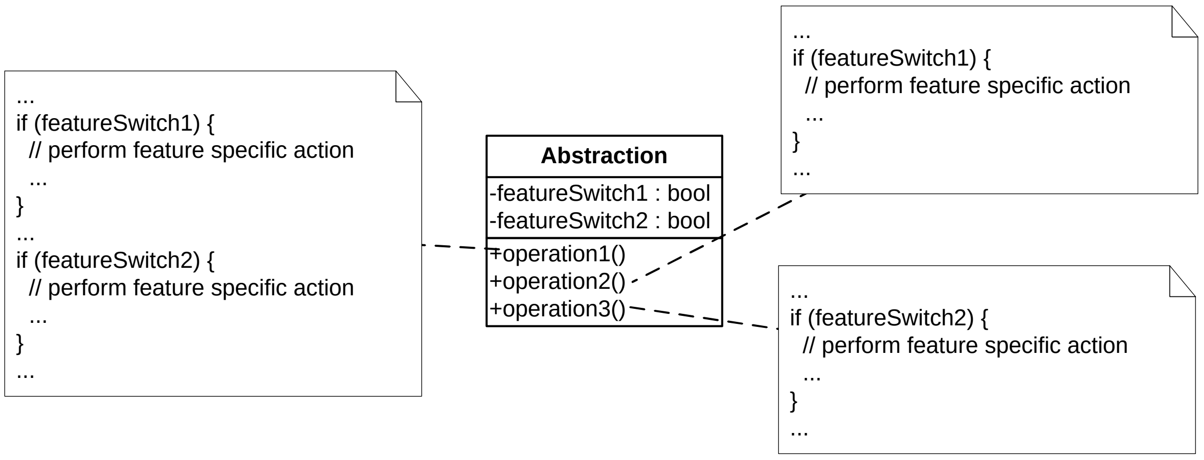


Figure A.14: Pathological structure for *embedded features*

check the value of the corresponding on/off switch, every time feature specific behavior might be desirable.

The pathological structure is hardly maintainable, because it mixes the implementation of the base abstraction with the implementations of the optional features. In addition, the ordering of feature specific actions, such as those in `operation1()` is fixed, and therefore impossible to alter at runtime. These aspects contradict the imperatives defined in the previous section.

A.5.5 Reference Structure

The reference structure for the design flaw "embedded features" corresponds to the decorator design pattern ([GHJV96]), and is illustrated in figure A.15. The original abstraction is extended with the class `AbstractDecorator`, which is a degenerate composite, in the sense that it composes exactly one instance of its parent. `AbstractDecorator` also serves as a base class to a number of classes, each capturing a unique feature. The abstract decorator (and therefore all concrete decorators) fully support the interface of the base abstraction. The default behavior of these methods is to transparently delegate to the contained instance of the base abstraction. Each feature can override one or more such operations, as well as extend the inherited interface, but should also call the default implementation. This allows clients to wrap several decorators around the object, in order to enable a combination of features. The wrapping order determines the logical layering of the features. Clients are responsible for ensuring that potentially invalid feature combinations or layering are avoided.

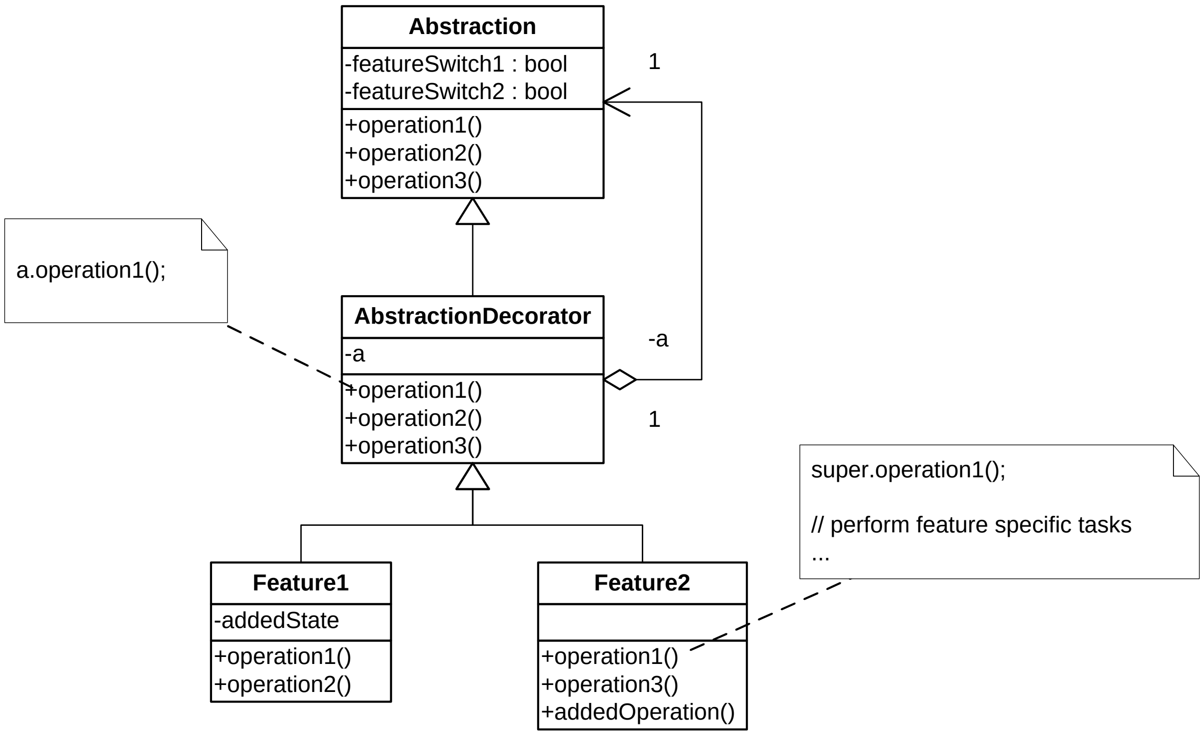


Figure A.15: Reference structure for *embedded features*

A.5.6 Diagnosis Strategy

Search space

All classes in the system.

Initial filter

The class defines at one attribute that appears to represent an optional feature of the class. In other words, the attribute has a boolean type, and it is written to either in a constructor, a method whose name contains “init”, “setup”, “configure” or “update” or from outside the class (either directly or through accessors), but not from other non-accessor methods of the class. In addition, the attribute is checked exclusively in simple conditional statements (if or if-else), in methods of the class. There are at least 2 different such attributes in a class.

Indicators

Indicator 1: *One or more of such attributes have in their names strings suggesting an ON/OFF switch. That is they either contain one of the strings “flag”, “feature” or “switch” or they begin with “has” or end with one of “On” / “Off” strings.*

Indicator 2: *Code that corresponds to a branch in one or more of the identified simple conditionals, acts as a filter or otherwise changes the return value of the method in which it resides, by either containing a return statement or by writing to a variable that is used as the return argument of the method.*

Indicator 3: *Almost all affected methods have a higher than average cyclomatic complexity. The average cyclomatic complexity should be project dependent.*

Context matching

Question 1: *It must be confirmed that the classes represents a valid abstraction in the design of the system.*

Question 2: *The maintainer must confirm that the class attributes identified by the initial filter represent on/off switches for optional features provided by the suspected class.*

Question 3: *The maintainer must confirm that new features are expected to be added in the future, changes are expected to occur to the existing features, or the need to dynamically change the logical layering of the features may arise.*

Question 4: *The public interface of the class is either not expected to suffer frequent changes, or changes are only expected in methods that are not related to any of the optional features.*

A.5.7 Reorganization Strategy

```
1: Let $C$ be the class containing the embedded features, and the so called enclosure type
   be the interface that declares all of the public methods needed by clients of $C$
2: if E is defined by a superclass or implicitly defined by $C$ then
3:    Extract an interface $I$ defining the enclosure type
4:    Make $C$ explicitly implement $I$
5: end if
6: Identify $F$, the set of optional features implemented in $C$
7: if $F$ contains more than one feature then
8:    Define a base decorator class $D$ (may be abstract), which implements $I$ and dele-
      gates to an internal instance of the type $C$
9:    Define concrete decorator classes for all features in $F$, as subclasses of D $D$
10: else
11:    Define a concrete decorator class, which implements the interface of the enclosure
       type and has delegation methods to an internal instance of type $C$
12: end if
13: if $C$ has subclasses that override optional features in $C$ then
```

```
14:     Move feature overriding code from the subclass into specializations of the corre-
        sponding concrete decorator classes
15: **end if**
16: **for all** features $f_i$ in $F$ **do**
17:     Move the corresponding code blocks out of the conditional statements from $C$ into
        the corresponding concrete decorator class. If there are methods that are called
        exclusively from such code blocks, move these methods to the decorator class as
        well
18:     Create appropriate constructors in the concrete decorator class
19: **end for**
20: Remove the configuration parameter from $C$
21: Adjust clients by replacing object parametrization with proper instantiation of the
    decorator and decorated classes. If several decorators need to be layered and there
    are invalid combinations provide and make use of factory methods in the decorated
    class
22: Adapt all clients that rely on the identity of the decorated object to eliminate this
    dependency // *This is necessary because decorated objects share the interface of the
    original object, but not its identity*
```

A.6 Duplicated Features

A.6.1 Description

To ensure the quality of a system, developers refer most of the time to object oriented principles and design patterns. One of these principles tells us that we should "Program to an interface, not an implementation" [GHJV96].

There are cases when classes have to be the specializations of other classes but, they also need to implement one common interface. This way a hierarchy is created that has as root the common interface and as implementors the specializations of other classes.

By applying the above mentioned principle, one might design the root in a way that could lead to code duplication in the specialized classes which implement it.

We have a case of "duplicated features", when in a situation such as the one described above, the root of the hierarchy defines a number of methods which have the same implementation in the specialized classes. Figure A.16 depicts an example instance of duplicated features in a browser system. The root interface *IURLManipulator* in the figure, provides the generic interface which defines common operations on manipulating URL's. The classes that implement this root element, *History* and *Bookmarks*, present specializations of two different graphical elements, *List* and *Table*. Two of the implemented methods (add(String), remove(String)) do the same thing in the same manner: store the given parameter, which represents an url, in a list.

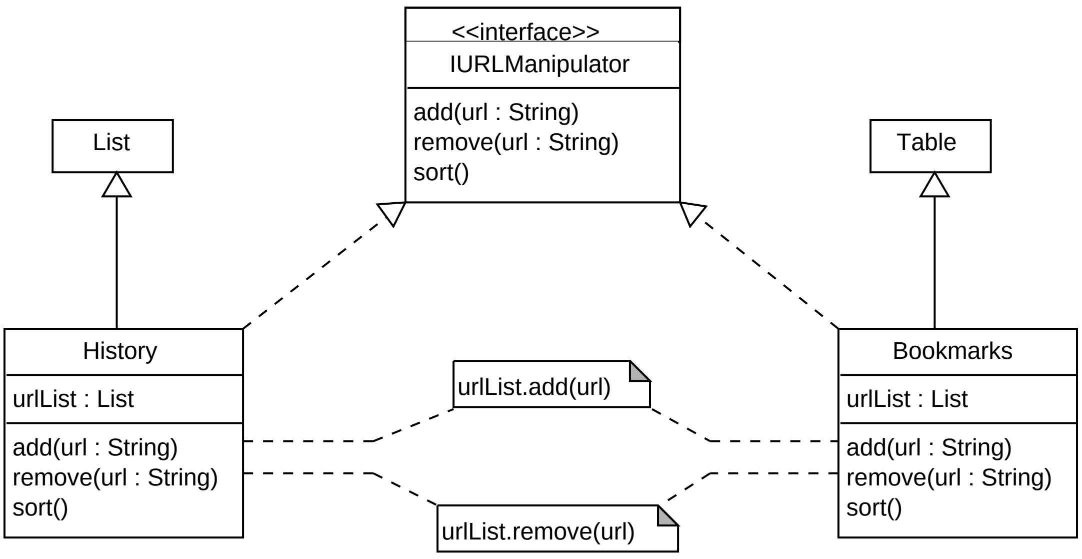


Figure A.16: An example of *duplicated features*

We can imagine that in the given situation other methods could be added to the root interface which could lead to other duplicated features. Besides that, other specialize classes that could be added to the hierarchy, would also duplicate these features. If a modification has to be done in the implementation of a feature, it will most likely have to be done in all specialized classes.

Therefore, a case of duplicated feature makes it hard to maintain and extend the hierarchy because of the duplications which are present each of its specialized classes.

A.6.2 Context

Design intent
The classes in the hierarchy must present the same features without duplicating them. The class, or hierarchy of classes represent a valid abstraction or specialization hierarchy in the application's design.

Strategic closure
You expect the number of features and the number of specialized classes in the hierarchy to change. Also, any feature's implementation might have to be changed in the future.

A.6.3 Imperatives

In order to maximize maintainability in the context described above, it is important for each specialization to present the features defined in the root interface but without duplicating their implementation. This is most naturally achieved with the use of containment and delegation relations.

A.6.4 Pathological Structure

As exemplified in figure A.17, each specialized class in the hierarchy duplicates the features defined in the hierarchy.

Although each class presents the requested features, they are actually duplicated. This way each class contains the implementation of common features, making the design fragment hard to change in the ways described in the context.

A.6.5 Reference Structure

The reference structure in the given context uses containment and delegation as the natural way to eliminate the duplicated implementations of the features.

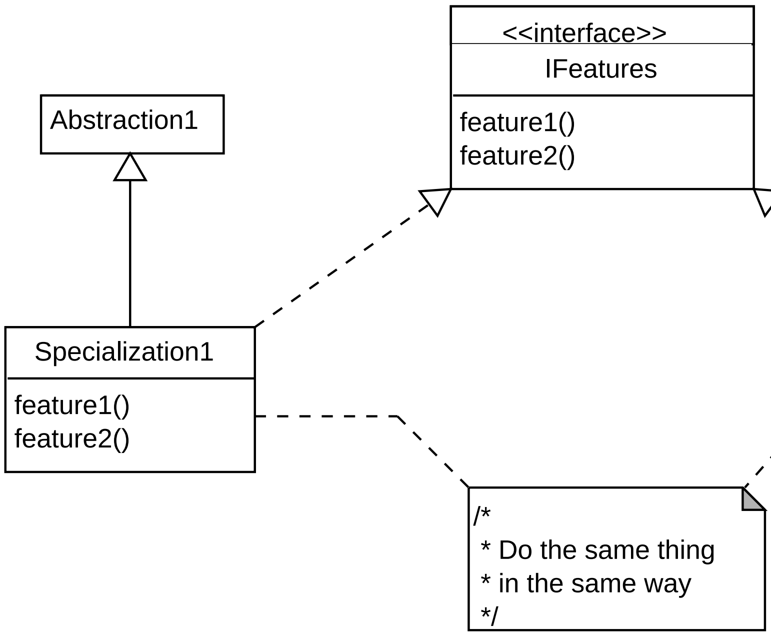


Figure A.17: Pathological structure for *duplicated features*

As depicted in figure A.18, the specialized classes do not contain anymore the implementations of their features. They have an attribute which holds a reference to an concrete instance on the interface which defines the hierarchy's features. Each operation representing a feature will be delegated to this attribute.

A.6.6 Diagnosis Strategy

Search space

All types in the system.

Initial filter

There are at least two classes which represent specializations of different supertypes but which have a common root. The root defines a number of methods that are implemented in all the above mentioned classes.

Indicators

Indicator 1: *Some of the methods defined in the root contain one of the words "add", "remove" or "get", suggesting storage features, thus indicating duplication.*

Indicator 2: *Implementation of some methods present code duplication in all the specializations which implement the common root.*

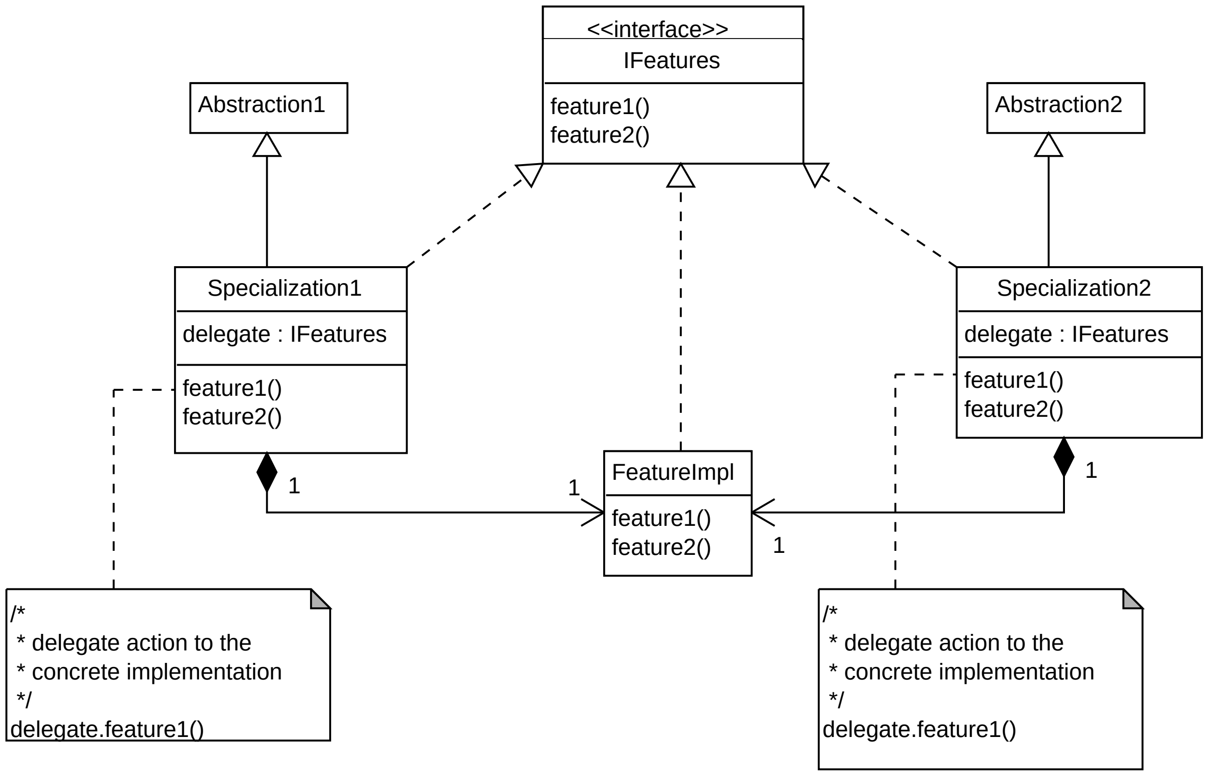


Figure A.18: Pathological structure for *duplicated features*

Indicator 3: *The affected methods suggest described in indicator 2 suggest independent features. That is, there is no call between any pair of such methods.*

Context matching

Question 1: *It must be confirmed that the affected classes represent valid specializations of different supertypes in the design of the system.*

Question 2: *It must be confirmed that the common root represents a valid abstraction in the design of the system.*

Question 3: *It must be confirmed that the affected methods represent valid features of the hierarchy and that their functionality is duplicate through each implementing class.*

Question 4: *It must be confirmed that the implementation details of some features or the number of specialized classes implementing the common root is expected to change.*

A.6.7 Reorganization Strategy

```
1: Identify each affected method of the common root of the hierarchy.
2: Create a class to implement the common root of the hierarchy and name it "FeatureImpl".
3: Move the implementation of each affected method in the "FeatureImpl" class. All the other methods will be given a default implementation or will be implemented as NOP's.
4: Create in each affected class an attribute of the common root's type and modify the constructor in order to initialize it, then modify each affected method's implementation to delegate it's responsibility to the corresponding method of the newly created attribute.
```

Bibliography

[Bae99] Holger Baer et al. The FAMOOS object–oriented reengineering handbook, 1999.

[Ciu01] Oliver Ciupke. *Problemidentifikation in objektorientierten Softwarestrukturen.* PhD thesis, Universität Karlsruhe, 2001.

[Fow99] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley Professional Computing Series. Addison-Wesley, 1996.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns.* Addison Wesley, 2004.

[Mar01] Radu Marinescu. The metrics meta-model. Technical report, Forschungszentrum Informatik (FZI), Karlsruhe, 2001.

[Mar03] Robert C. Martin. *Agile Software Development. Principles, Patterns and Practices.* Prentice Hall, 2003.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice Hall, 1988.

[Par94] David Lorge Parnas. Software aging. Technical report, Communications Research Laboratory, Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario, Canada L8S 4K1, 1994.

[Pre00] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill Series in Computer Science. McGraw-Hill, 2000.

[Rie96] Arthur J. Riel. *Object–Oriented Design Heuristics.* Addison–Wesley, first edition, 1996.

[Tri08] Adrian Trifu. *Towards Automated Restructuring of Object Oriented Systems.* University of Karlsruhe, 2008.