Masters thesis

Facet-Oriented Program Design

by

Mikael Amborn

LITH-IDA-EX–04/047–SE

May 17, 2004

Linköpings universitet
Institutionen för datavetenskap

Masters thesis

Facet-Oriented Program Design

by **Mikael Amborn**

LiTH-IDA-EX–04/047–SE

Supervisor: **Prof. Uwe Assmann**
Department of Computer and Information Science
at Linköpings Universitet

Examiner: **Prof. Uwe Assman**
Department of Computer and Information Science
at Linköpings universitet

Abstract

This thesis examines the concept of separation of concerns in software engineering. By taking inspiration from the field of information classification and the technique of faceted classification it suggest a new method to separate concerns in software. The method, which we call facet-oriented programming, is described and various example of its use is shown. Possible support for facets in a programming language is discussed and the discussed features are then implemented in a compiler for the Pike programming language.

Keywords: software design, separation of concerns, faceted classification, Pike, multiple inheritance

Contents

Chapter 1

Introduction

This chapter gives an introduction to the thesis, explains its purpose and goal, and provides an overview of the rest of the document.

1.1 Purpose of This Thesis

A common way to classify information is by using an enumerative scheme. In such a scheme we start by identifying the main diciplines to be covered by the scheme and allocating each dicipline a class, e.g. 1 = dog, 2 = cat, 3 = horse, etc. Then each dicipline is divided into subclasses, e.g. 1.1 = labrador, 1.2 = poodle etc. This process of subdivision is continued until an appropriate level of specificity has been achieved. The object is to provide one place, and one place only, for each subject. For example, the Dewey Decimal Classification [3] classifies ”Philosophy and Psychology” as class 100, which is further broken down into 100: Philosophy, 110: Metaphysics, 120: Epistemology, etc. Many enumerative schemes are thus also hierarchical.

Enumerative classifications are essentially top-down methods of analysis: you start at the top of the hierarchy and work your way down until you find a suitable heading or term that describes or classifies your text.

Faceted classification is a completly different system of classification, and is unlike the enumerative system a bottom-up system. That is, it starts with the object being classified and use that as the basis for the descriptions. A faceted classification model consists of a number of facets, or dimensions, with terms defining the dimension. To classify an object we select the term from each facet that best describes our object. The facets can be seen as characteristics or attributes of the object. As an example[1] say we want to classify a ball. We might describe it as “round”, “soft”, “blue”, “large” and “heavy”. These sets of terms can be generalized to form a set of mutually exclusive characteristics (facets):

- Shape
- Texture
- Color
- Size
- Weight

With these facets we can describe any ball or any number of other objects. Each of the above facets will contain terms that we use when we want to classify our object. The facets can easily be extended with new terms as needed. So we do not have to predict all possible terms that might be needed from every facet. The system is thus easily extended, unlike an enumerative system, where it is almost impossible to create new classes without affecting earlier classifications.

The purpose of this thesis is to examine how facets can be used in connection with Object-Oriented(OO) programming. If we let classes in OO-programming play the roles of both the terms in the facets and of the information to classify, multiple inheritance can be seen as a classification process, where a child class is classified by its parent classes. This thesis will discuss this issue in some detail, look at some examples where it might

[1]From: `http://www.contextualanalysis.com/pub_usingfacets.php4`

be usefull and also look at what kind of support a compiler can provide when programming with facets.

1.1.1 Typographic conventions

The following typographic conventions will be followed throughout this report:

Program code: will be set in this font, with keywords such as **int** set in bold, both in longer listings of code and when referring to specific elements of program code in the main text, e.g. to function names, class names, etc.

New concepts: will be set in *italics* when they are introduced. Italics will also be used to emphazise words and to give titles of books.

File names: will be set in `tele type` font.

1.2 Report Outline

This section describes the outline of the rest of the report.

Chapter 2 looks at separation of concerns in software engineering and some common techniques used to achive it.

Chapter 3 describes the history of facets and their field of use today.

Chapter 4 discusses when and in which ways facets can be used in software design.

Chapter 5 examines ways to implement facets in program code, both with multiple inheritance and with a variant of the bride pattern.

Chapter 6 discusses how the programming language Pike has been extended to support programming with facets.

Chapter 7 looks at some examples of facet-oriented designs.

Chapter 8 looks at and discusses the results of this thesis.

Appendix A lists the code added to the Pike compiler.

Appendix B lists the code for one of the example applications from chapter 7.

Chapter 2

Separation of Concerns

This chapter looks at the concept of separation of concerns in software engineering.

2.1 Introduction

A core concept in software engineering is *separation of concerns* [11]. This refers to the process of identifying and encapsulating the parts of a system that are related to a particular concept or task. Relevant concepts or tasks can be security, data structures, logging, specific algorithms, etc. By encapsulating concerns we reduce the program complexity and improve comprehensibility. We also limit the impact of change to the program and facilitate reuse, e.g. it is easier to reuse an algorithm from a program if the algorithm is encapsulated in its own function. So dividing a program into function, i.e. procedural programming, is a way to separate functional concerns. In the same way Object-Oriented programming is an example of a method to separate data concerns in a system into classes.

Separation of concerns is closely related to the divide and conquer method, which aims at breaking down a large complex problem into smaller more easily solved subproblems. By solving these subproblems and merging their results we get the answer to the initial problem.

To achieve a clean separation of concerns in a system different techniques at different levels of the program structure have been developed. In the rest of the chapter we will look at some of the techniques that has emerged in the last couple of years.

2.2 Class and Function Based Methods

The following methods works on single functions and/or classes.

2.2.1 Aspect Oriented Programming

As we chose to modularize a system according to a given concern other concerns tends to get intertwined into the code. E.g. in object-oriented design functional concerns, such as logging or security, often crosscut several of the classes. So if we for example want to enforce a new security policy in our object-oriented application we will have to first find all places where security code is present and then update them in a consistent way. If the security code is well separated from the rest of the code we might be able to encapsulate it in a separate class. We would then only need to replace/modify that class to implement a new security policy without lots of invasive changes to the rest of the code. But if the security code is deeply intertwined with the surrounding code separating it might be very hard or even impossible.

One way to solve the problem of cross cutting concerns in systems is *aspect-oriented programming* (AOP) [8]. With this method we can separate for example security code that is cross cutting several functions and/or classes while keeping the original structure of the code. The security code is separated from the main code into an aspect. By specifying where the aspect-code should be inserted the compiler or a preprocessor can weave it back into place.

AspectJ is an extension to the Java language that adds support for aspects.

The example code below shows a class Circle with a constructor and two functions. Each function starts by printing its signature.

```
public class Circle {
  private int radius;
  public Circle(double r) {
    System.out.println (“ Circle(double)”);
    radius = r;
  }

  public double perimeter() {
    System.out.println (“ perimeter()’’);
    return 2 ∗ Math.PI ∗ r;
  }

  public double area() {
    System.out.println (“ area()’’);
    return 2 ∗ Math.PI ∗ r ∗ r;
  }

}
```

The printouts in the three functions are almost identical and can be moved to an aspect, the resulting code is shown below where the new aspect is named Logging.

```
aspect Logging {
  pointcut CircleFunctions(): call(public ∗ Circle .∗(...));

  before (): CircleFunctions() {
    System.out.println(thisJoinPointStaticPart.getSignature());
  }
}

public class Circle {
  private int radius;
  public Circle(double r) {
    radius = r;
  }

  public double perimeter() {
    return 2 ∗ Math.PI ∗ r;
  }

  public double area() {
    return 2 * Math.PI * r * r;
  }

}
```

The keyword pointcut in the aspect Logging gives a name to places in the main code where we want the aspect to do something. We give the pointcut the name CircleFunctions and specify it to match all public functions in the Circle class. The before keyword then specifies that we want to do something before the places specified in the following pointcut name. Here we want to print the signature of the functions that is being called, which is accessed with the statement thisJoinPointStaticPart.getSignature();.

As can be seen from the example above the code becomes cleaner with aspects and if we want to change the logging printouts we only need to modify the aspect Logging and not each of the affected functions. The example is very simple but imagine a system with several hundred classes each with its own functions. It would be extremely tedious to edit each of these functions when we want to change the logging printouts. For other example uses of aspects and AspectJ visit `http://eclipse.org/aspectj`.

AOP thus allows us to separate our code according to several concerns simultaneously. Concerns that ordinarily would cross cut each other and be impossible to separate cleanly.

But we can still run into problem with the separation of concerns, because, different concerns may be relevant depending on the current stage in the software life cycle and the software engineers role in the project. A concern that was natural to separate into an aspect during the initial stage of the software life cycle may later in itself prove to contain cross cutting concerns with other aspects and the main code that we now want to separate. It is impossible to define aspects of aspects with AOP, aspects can only be weaved together with the main code not with other aspects.

Event-based Aspect-oriented programming (EAOP) is a way to solve the problem with creating aspects of aspects. In standard AOP we specify where in the code each aspect should be inserted, but in EAOP we instead get to specify at what events in the program execution each aspect should be “inserted”. We can for example specify that an aspect should be run the fourth time a certain function is called or each time some other aspect is running, see `http://www.emn.fr/x-info/eaop/`. Another way to solve the problem is with hyperspace programming which will be discussed in the next section.

2.2.2 Hyperspace Programming

Hyperspace programming [10] is being developed by IBM and stems from *subject-oriented programming*(SOP) [6]. Subject-oriented programming focuses on the fact that clients of a class can have different interest or views on the class. For example the clients of a tree class can see the tree as a living being or as potential material that can be transformed into furniture. These different views can either be incorporated into the tree, which will get unmanageable if the number of clients of the class is big, or stored with the clients of the class, loosing the benefits of encapsulation and polymorphism for the tree class. Subject-oriented programming solves this problem by introducing the concept of subject. A subject is a view on a class, or an extension to a class. So for a bird client of the tree class we can add new functionality to the tree without modifying the original source code for the tree class. All the subjects of a class is then combined into a whole. In this way we can add new features to an existing class without any invasive changes to the class, and the main class can remain small and manageable. Read more at the SOP homepage [19].

As an extension to SOP IBM is now developing hyperspace programming. Hyperspace programming is IBM's implementation of multi dimensional separation of concerns, it allows us to create aspects of aspects and to separate concerns in different dimensions simultaneously. The research project at IBM strives to provide full support for multi-dimensional separation of concerns which will allow on-demand remodularization of systems. Thus allowing a developer to choose at any time the best modularization, based on any or all of the concerns, for the development task at hand. IBM's researchers have also developed a tool, called Hyper/J, which adds support for hyperspaces to the Java programming language [7].

2.2.3 Composition-Filters

“Composition-Filters is an aspect-oriented programming technique where different aspects are expressed in Filters as declarative and orthogonal message transformation specifications.“ [1] *Composition-Filters* works like wrappers around existing classes. All messages to and from the class pass through the composition filters which can modify the message as they see fit, see figure 2.1.

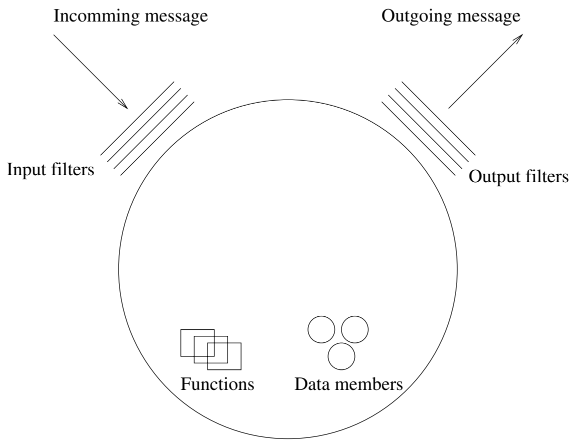


Figure 2.1: An object with ingoing and outgoing composition filters.

In this way the class functionality can be extended in a modular way without modifying the original code. An arbitrary number of filters can be added to a class as the filters are orthogonal to each other [2]. Each filter decides what to do with the messages they receive and if to pass it on in the chain or not.

2.3 System Methods

In the following sections we look at methods that work on the system level and include multiple classes. They are therefore not as fine grained in

[1]`http://trese.cs.utwente.nl/composition_filters/filter_aspects.htm`

their separation of concerns as the above but rather help in separating concerns when developing object-oriented systems, or suggest ways to design to minimize the amount of cross cutting concerns in the system.

2.3.1 Role Modeling

Role modeling was developed by Trygve Reenskaug and described in his book *Working with objects* [15]. For a more concise introduction to role modeling see Riehle and Gross' paper on role model based framework design [16]. In an an object-oriented system a class can play different roles depending on the client and the state of the class-object. E.g. a class can both be an observer in the Observer design pattern and a graphical element in a drawing program. Reenskaug therefore suggested that instead of modeling classes directly, we should begin by modeling the different roles needed in the system. These roles can then be mapped to classes in a class diagram. By modeling roles instead of classes we get a better understanding of the system and the object collaborations. The mapping from roles to classes can also be done in different ways, so the role model is a more general model of our system than the class diagram.

A role model consists of role types, that describes the view one object holds of another object. An object which conforms to a given role type, is said to play that role. At any given time, an object may act according to several different role types. Thus, different clients may have different distinct views on an object. Also, different objects may provide behavior specified by a common role type.

We might for example have an application where we want to model a figure hierarchy in a drawing program. The model will consist of figure objects, both simple ones like line, and composite ones that consists of other figure objects. The model will also have a root object, that acts as the root of the figure hierarchy and contains some extra management functionality. Figure 2.2 shows a possible role model for the figure hierarchy. In the figure, role types are qualified by a role model printed in parentheses below the role type name. A role model defines a namespace for the role type that it defines, so that the role type Parent from the Figure role model is fully qualified as Figure.Parent.

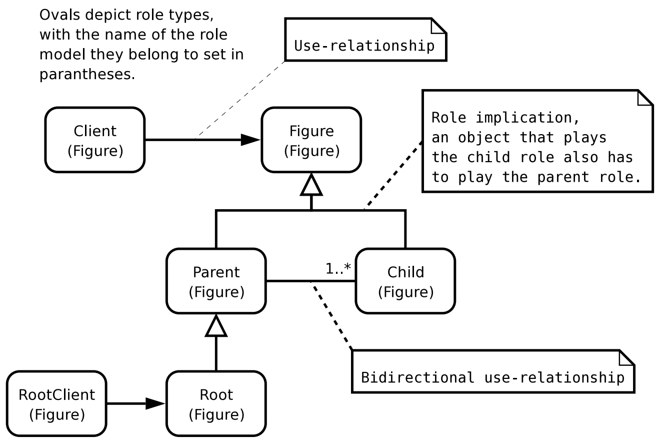

Ovals depict role types, with the name of the role model they belong to set in parantheses.
Use-relationship
Role implication, an object that plays the child role also has to play the parent role.
Client (Figure)
Figure (Figure)
Parent (Figure)
1..*
Child (Figure)
Bidirectional use-relationship
RootClient (Figure)
Root (Figure)

Figure 2.2: Role model for the figure hierarchy.

Our application will of course consist of several role models, and before mapping these role models to a class diagram they have to be merged into a single combined role model. In this combined role model we link the role types of the different role models to each other. We might for example have a role model FigureObserver that models the Observer design pattern, i.e. we want to be able to observe figure elements. This model must then be linked with our Figure role model, so that objects playing the Figure.Figure role type also have to play the FigureObserver.Subject role type, and vice versa.

When we have finished with the role modeling the roles are mapped to classes, this mapping can be done in different ways depending on our needs. From our Figure role model we can for example create three classes: Figure, CompositeFigure and RootFigure, the Figure.Client and Figure.Rootclient role models will be played by some other part of the drawing application that uses our figure classes. The resulting class diagram, with the roles played by each class placed on top of the classes, is shown in figure 2.3.

Because role models focuse on one particular collaboration purpose between objects, they provide excellent separation of concerns. Compared to traditional class diagrams that intertwine all different object collaboration

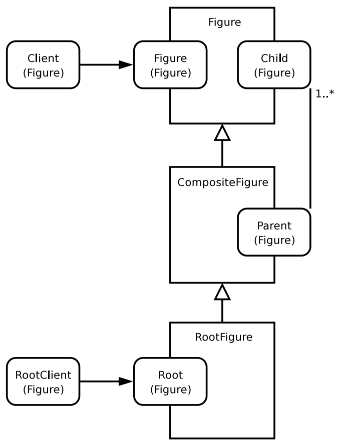


Figure 2.3: Class-role diagram for figure hierarchy.

aspects.

2.3.2 Layered Systems

Designing a system by dividing it into layers has long been a popular method to ease development and get cleaner code. Operating systems, for example, have for a long time been developed in a layered fashion, starting with the THE operating system which had six layers:

layer 5: user programs

layer 4: buffering for input and output

layer 3: operator-console device driver

layer 2: memory management

layer 1: CPU scheduling

layer 0: hardware

Three other examples of layered models are:

The three tier application model that is used extensively in web applications, usually with the layers: presentation, business logic and data handling.

The ISO/OSI network model, the standard model for networking protocols and distributed applications.

Layered frameworks, i.e. a framework built in layers. These are becoming more and more important as the need for reuse in software projects increases.

So how is layered systems coupled to separation of concerns? The layering separates the concerns between the layers. E.g. in the three tier application model the bottom layer is responsible for data handling and storage, the middle layer for application logic and the top layer for presentation and user interaction. By following this model when designing our web-application we will get a coarse grained separation of concerns in our application. If we wish we can then use one of the techniques described in the previous section to further separate concerns in each layer.

Chapter 3

Faceted Classification

This chapter looks at the history of faceted classification and its uses today.

3.1 History

The term faceted classification was introduced in the 1950's in two separate fields. Ranganathan used the term to denote aspects or viewpoints in library classifications systems [14] and Guttman used the term for designing sociological surveys [5]. The problem Ranganathan wanted to address was that books in a library could be classified differently based on different purposes. The standard way to classify books was with a enumerative method such as Dewey Decimal. In a enumerative scheme all possible classes are predefined. The librarian selects the class that best fits the attributes of the new item by traversing the classification hieararchy. A part of the Dewey Decimal hierarchy can be seen in figure 3.1.

The problem with this system is that it requires experties in both the area of the book to classify and the classification system. There might also be ambiguity in which class the book belongs. A book about both computer science and military engineering will have to be classified under one of 004

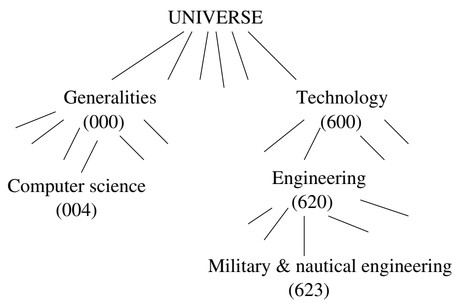


Figure 3.1: Part of the Dewey Decimal hierarchy.

or 623, neither of which is perfect.

Ranganathan introduced a faceted classification to solve the problems with the enumerative methods. A facet model consists of different facets and terms that are used to classify the information. To classify books in a library we can for example have the following facets and terms, among others:

Facet	Example terms
Type	biology, novel, poetry
Audience	childern, young, adult
Format	text, audio, video
Environment	sea, land, air

To classify a book we select a term from every facet, if a facet does not apply to the book we skip that facet. So a childrens book describing life in the Balltic sea could be classified with the terms biology, children, text, sea in the above order of the facets.

The advantage with this model over the enumerative one is that we can create all combinations of terms from the facets that we may need. With a faceted model there is no predefined group that we have to try to match our book against, we just select the most appropriate term from each facet. Faceted classification is today widely used in libraries all over the world.

3.2 Information Classification

Faceted classification can be used when classifying any kind of information, not only books in a library. The FacetMap system[1] for example uses facetted classification to classify information on a web site. With this system the user can browse trough the site starting with any one of the available facets. For example if you have a site with information about different locations you have visited with reviews of restaurants, hotels, bars and clubs at each location you could let the user browse the information by location or by restuarants that you gave 5 stars. So if the user is looking for the best restaurants you have visited she can browse your site by restaurants without the need to search trough each individual location. FacetMap builds on the XML standard XFML, eXchangeable Faceted Metadata Language, which is: “an open XML format for publishing and sharing hierarchical faceted metadata and indexing efforts”, see `http://xfml.org`.

Other fields where faceted classifiaction has been used or suggested to be used include:

- In artificiall intelligence, information or knowledge needs to be classified in a good way. To do so Uta Priss sugests a faceted knowledge representation, see [13].
- As an approach to multi-perspective modeling in requirements engineering, see [9].
- In the classification of business organizations, see `http://www.kmconnection.com/DOC100100.htm#business_vs_lib_sci`.

3.3 Software Reuse

Faceted classification can also used to classify components in a software resuse library. The library consists of software components, and the problem is how to find a suitable component from the library. Prieto-Diaz

[1]`http://facetmap.com`

suggested a faceted classification of the components [12], with six facets: function (operation performed), object (type of data object on which the operation is performed), medium (larger data structure in which the data object is located), system type (type of subsystem for which the component was designed), functional area (application dependent activities), and setting (application domain).

The programmer searching for a component that fullfills certain requirements selects terms from the facets and gets a list of matching components back. With the use of a conceptual graph of the terms, that measures how closely related terms are, similar terms can be automatically selected if the exact query does not match any components. Thus enabeling inexact or relaxed searching among the components in the library. Another benefit of using a faceted classification for the components is the possibility of adding new terms to the facets at any time. This is an important aspect when classifying components in a new and rapidly developing field such as computing, see [1]. Prieto-Diaz reports a four-fold improvement in precision/recall ratio when using a faceted classification compared to a retrival system not organized by a classification scheme.

Chapter 4

A New Approach to Faceted Classification

As mentioned in chapter 1, facets are a good way to classify information in a rapidly changing field of knowledge. It is easy to add new terms in the facets without affecting earlier classifications. To be easy to extend is also a desireable attribute of computer programs. So if we can organize our class structure into facets this should help in making the program easier to extend.

By letting classes in a object-oriented system represent both terms in the facets and the information to classify we can map the facet concept to object-oriented programming. We will refer to a class in a facet as a *facet-class* and to a class inheriting from one class in every facet but not belonging to the facet as a *product-class*. In faceted information classification each object is classified with one term from each facet and here the object/product-class will be associated with one class from each facet using multiple inheritance.

By inheriting from a facet we will mean inheriting from a facet-class. We will further refer to a software design using facets as a facet-oriented design.

When using facets in object-oriented programming we can view the facets in, at least, two different ways:

Orthogonal facets are similar to the use of facets in faceted information classification. The facets represents different aspects of the system.

Sequential facets are similar to layered system, see section 2.3.2. The facets represent layers that information flows through in a sequential order.

Using facets in object-oriented programming is also related to the process of separation of concerns. Each facet can be said to encapsulate a certain concern, because each facet deals with one dimension or aspect of the product-class.

4.1 Orthogonal Facets

We use the term orthogonal facets, in connection to software design, to mean facets as they have been described earlier in this report. That is, each facet describes one aspect of some area of interest. However, unlike when used for information classifications, the facets will now contain classes with functionality that the product-class will inherit.

For example, say that we need to model animals in our program. We could then use the model shown in figure 4.1, where we use UML-packages to represent the facets. Each of the facets in the figure contains classes with functionality that defines some aspect of the animal. E.g. the ByMovement facet contains four classes that defines different kinds of movement. The facets have been chosen to be orthogonal and to best separate animals into different classes.

In the figure we have four facets that describes our animals from different perspectives. Each facet participates with some functionality that is specific for that facet. As this is just an example there is not much content in each facet-class, but the idea should be clear. When we want to add an animal to the system we choose the class from each facet that best describes the

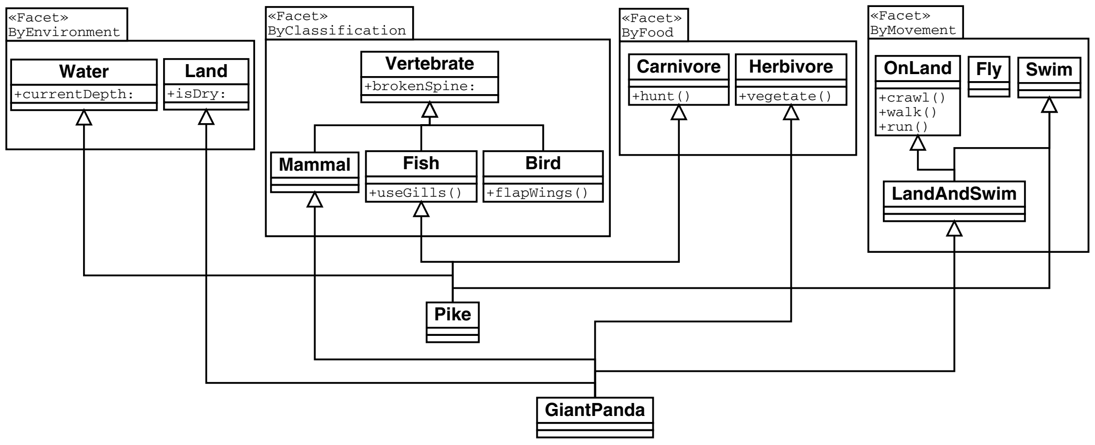


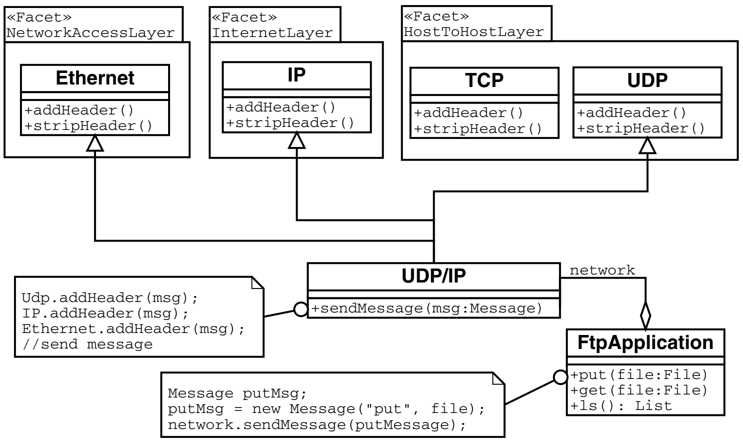
Figure 4.1: Facet-oriented design for animals.

animal. When, for example, adding the Pike class we choose Water from the ByEnvironment facet as it lives in water, Fish from the ByClassification facet, Carnivore from the ByFood facet, and Swim from the ByMovemet facet. With this design we can encapsulate all functionality that has to do with for example swiming in one class. As you can see from the figure we can also combine classes in a facet to form useful combinations such as LandAndSwim for animals that can move both in water and on land. In this way we keep to the restriction that each product-class only uses one class from each facet.

Dividing the class hierarchy into separate dimensions in this way means that we factor our design. If we where to implement it as one large hierarchy we would either need to implement the functionality from every facet-class in all product-classes that needs it. Or we would need to create classes for all combinations of facet-classes that we would need, e.g. for carnivore fishes we would need a class with the contents from the Water, Mammal, Carnivore, and LandAndSwim classes. So, in the worst case we would need $d_1 * d_2 * \cdots * d_n$ different classes as combinations of the facet-classes, where d_x is the number of classes in the class hierarchy of dimension x. But with a faceted class hierarchies we only need $d_1 + d_2 + \cdots + d_n$ classes.

4.2 Sequential Facets

In the three-tier application model we typically have layers for data handling, application logic, and presentation. Even though the layers can be independant they are not really orthogonal, because they need the functionality from the other layers to function. Rather than describing different aspects they represent a workflow in the application. Data typically goes from the data layer trough the application logic and to the presentation layer and back again. The layers work on the same data but in different ways. Compared with orthogonal facets where the facets do not have to have anything at all to do with each other. For such layered systems with independant layers we use the term sequential facets because the facets typically work on the same data in a sequential way. Figure 4.2 shows an example design of a network stack with sequential facets.



```
Udp.addHeader(msg);
IP.addHeader(msg);
Ethernet.addHeader(msg);
//send message
```

```
Message putMsg;
putMsg = new Message("put", file);
network.sendMessage(putMessage);
```

Figure 4.2: Facet-oriented design for a network stack.

In this design each facet processes the data to be sent over the network by adding or removing header information. So the facet classes clearly works on the same data but they are still independant of each other. Of course the UDP class depends on the IP and Ethernet classes for the message to be sent over the network but it does not care how this is done.

4.3 With Role Modeling

If we use role modeling when designing our software system it will be easier to recognice when it is possible to use a facet-oriented design in our class model. If the roles can be split into independant groups there is a good chance that we can use facets.

As an example, suppose we want to design an application that will model a software company with programmers, designers and managers. We want the application to model the degree of experience of each employee and their respective field of expertise, e.g. Unix system programming, Windows graphics programming, etc. These requirements could lead to the combined role model in figure 4.3, see figure 2.2 in section 2.3.1 for explanation of the syntax.

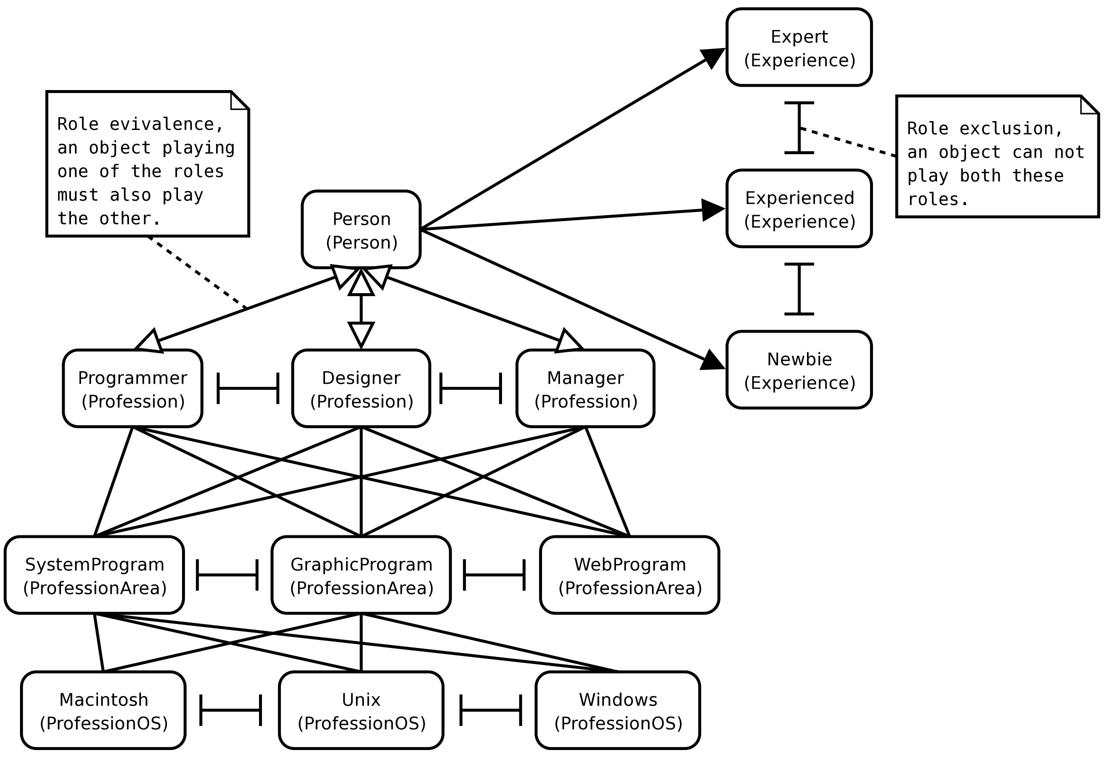

Expert (Experience)
Role evivalence, an object playing one of the roles must also play the other.
Role exclusion, an object can not play both these roles.
Person (Person)
Experienced (Experience)
Newbie (Experience)
Programmer (Profession)
Designer (Profession)
Manager (Profession)
SystemProgram (ProfessionArea)
GraphicProgram (ProfessionArea)
WebProgram (ProfessionArea)
Macintosh (ProfessionOS)
Unix (ProfessionOS)
Windows (ProfessionOS)

Figure 4.3: Role model for software company example.

From the figure we can see that all role types in the Profession role model can use any one of the three role types in the ProfessionArea role model, i.e. regardless of profession the employees are specilized in one area. Furthermore, an employee can only have one profession and be specilized in one area. All role types in the ProfessionArea role models are linked to an operating system in the ProfessionOS role model, except for WebProgram, because web programming is basicly the same across all platforms.

From the figure we can see that we can achive a faceted model by mapping every role type to its own class. We then create a facet for every role model and place the corresponding classes in them, except for the Person.Person role type that we place as the top class in the Profession facet. Figure 4.4 shows the resulting class diagram. Each role model is mapped to a facet and each role types to a facet-class.

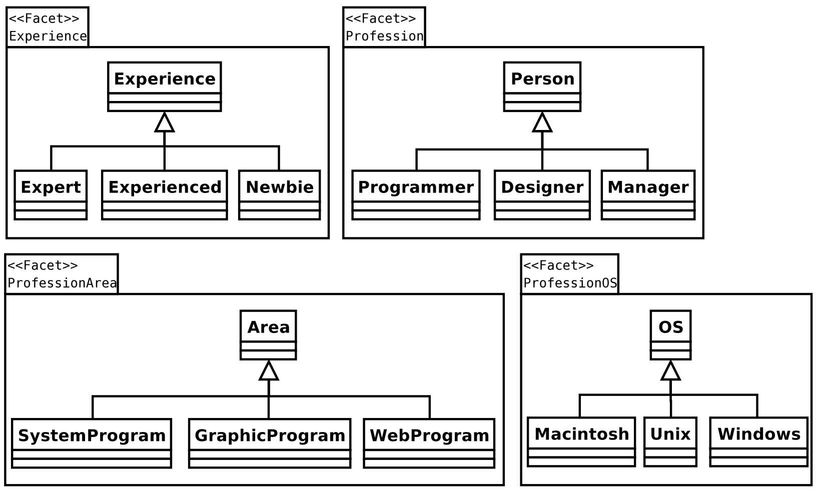


Figure 4.4: Class model for software company example.

An alternative mapping using a faceted class model could have joined several role models into a singel facet. For example the ProfessionArea and the ProfessionOS role models could have been merged into the same facet. We could also have mapped the role model to a class model without a faceted structure. An example of such a mapping is shown in figure 4.5.

As can be seen from the figure it is possible to map role types that have a

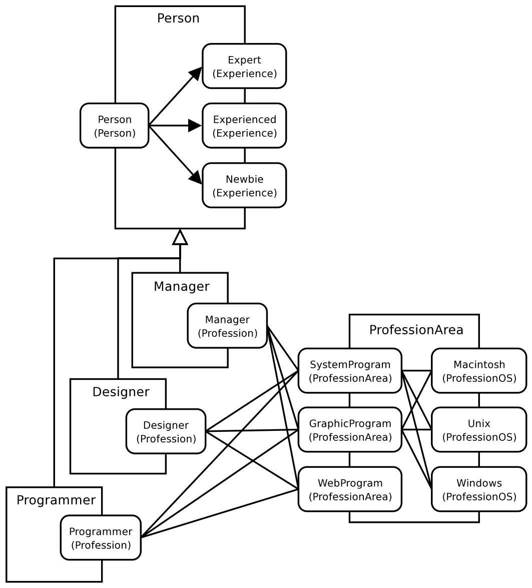


Figure 4.5: Alternative class-role model for software company example.

role exclusion constraint between them to the same class. The constraint only says that *objects* are not allowed to play the roles at the same time. So an object of the Person class, or one of its subclasses, can at any given time only play one of the roles from the Experience role model.

The advantages of using a faceted class model in this example is that it makes it very easy to extend the model. We can add new classes to any one of the facets without affecting earlier product-classes, or having to recompile them. However if we want to add, for example, a new area of expertise to the second class model, we would have to modify the ProfessionArea class. This would also mean that we would have to recompile all the classes that uses the ProfessionArea class as well.

Chapter 5

Programming with Facets

In this chapter we take a look at different ways to implement a facet-oriented design in program code.

5.1 Introduction

The UML diagrams presented so far in the thesis has shown product-classes inheriting from the facet-classes. This can be directly implemented in program code, but we can also implement the facets with an *n*-Bridge pattern. If the implementation language does not support multiple inheritance we can use the n-Bridge pattern instead. The following chapters will look at both of these methods.

The term *facets* in connection with programming is also used by the Systems Research Group at the University of Hong Kong, `www.csis.hku.hk/~clwang/projects/sparkle.html`, for what they call Facet-based programming (FBP). In FBP a facet is similar to a class in an OO-language but without internal state. In this thesis the term is, however, not used in that sense but in the same way as previously in the report.

5.2 Implementation with Bridge Pattern

The Bridge pattern is described in the book *Design Patterns* [4], henceforth referred to as the GoF book, as a way to separate an abstraction from its implementation. Suppose that we have an application that draws windows on the screen implemented for a X-windows system. Part of the class diagram for the system is shown in figure 5.1. The code for the different window types is mixed with the code specific for drawing the respective windows on the X-windows system.

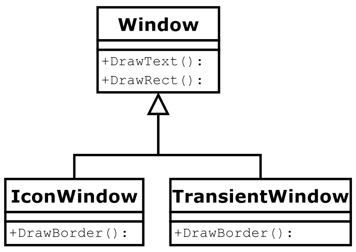


Figure 5.1: Class diagram for the windows application example.

If we want to port the application to another windowing system such as Windows XP, we would need to create new classes for each window type with the X-window system specific code replaced with Windows XP specific code. The class structure for the resulting system is shown in figure 5.2.

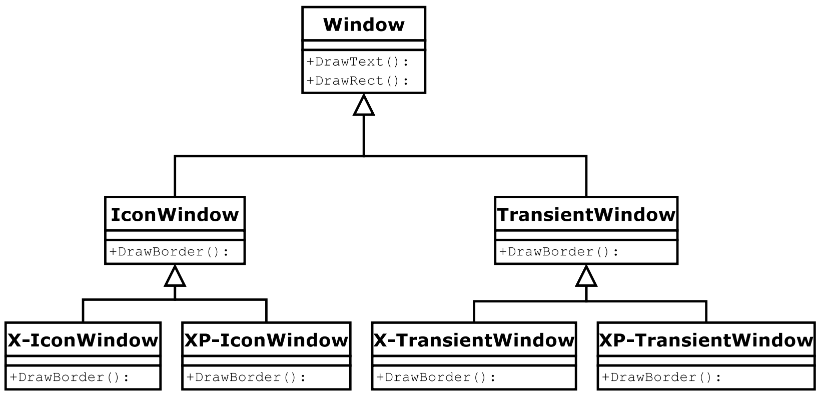


Figure 5.2: Class diagram for the windows application for two systems.

When we want to add support for additional systems the class hierarchy will become increasingly complex. This is what the authors of the book *Object-oriented Modeling and Design* [17], calls nested generalization. The Bridge pattern suggests that we separates the window abstraction from the implementation platform into independant class hierarchies, se figure 5.3. The abstraction hierarchy consists of different window types such as icon window and transient window. While the implementation hierarchy consists of different plattforms we want to use the application on, e.g. X-windows and IBM's Presentation Manager. A client class of the Bridge pattern then selects a window to create from the abstraction hierarchy and a platfrom to show it on from the implementation hierarchy. This allows us to add new windows without modifying the implementation classes and to add new plattforms without modifying the window classes.

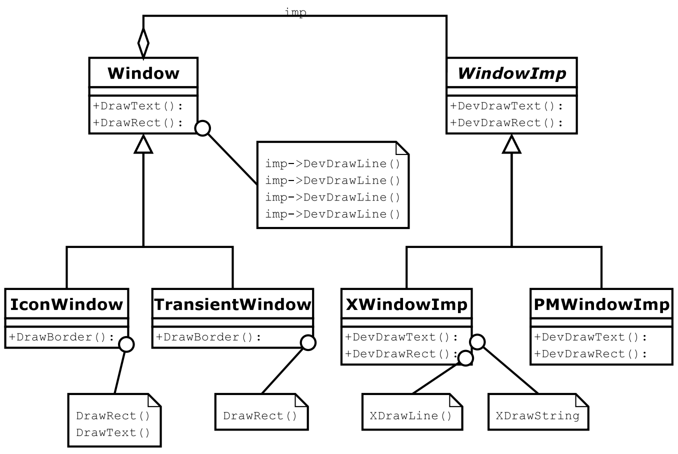


Figure 5.3: Structure of the Bridge design pattern.

The motivation for the pattern in the GoF book is for the programmer to be able to create objects of the abstraction, e.g. windows and icons in a graphical user interface, without having to concider how the abstraction are implemented, e.g. in which graphical environment the windows and icons will be drawn. But the Bridge pattern can also be seen as a facet-oriented design with two facets; Abstraction and Implementor.

The *n*-Bridge pattern augments the Bridge pattern with additional hierarchies like the Abstraction and Implementor as shown in figure 5.4.

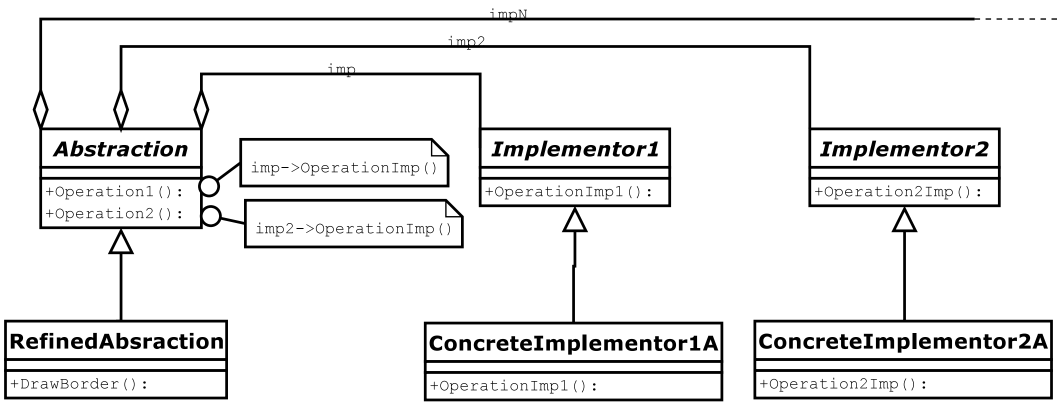


Figure 5.4: Structure of the n-Bridge design pattern.

By letting one of the facets in a facet-oriented design serve the role that the Abstraction class does in figure 5.4 we can implement the design with the n-Bridge pattern. We could also let a separate class outside the facets play this role.

The consequences of implementeing a faceted design with the n-Bridge pattern are:

- The chosen class from each facet can be changed dynamically at run-time.
- Classes from the different facets can be combined arbitrarily, i.e. there is no restrictions on what facet-classes we use together.
- Object schizophrenia, the choosen classes from each facet are together supposed to describe one physical object, such as a window on the screen. But with the n-Bridge pattern it is represented with several objects from the different facets, each with its own identity. This leads to object schizophrenia which can cause the following problems:

  Broken Delegation Consider what happens when an object A implements an operation, op1, by passing it to another object B, with a method op1. If the object B makes an operation call on itself from op1 to op2 it finds the definition of the implementation in itself, and not in the class A. It gets the wrong semantics if the method implementation specified in A, of op2, has been overridden in B.

Doppelgängers Consider a component of the system which maintains a set of objects which register and de-register for any of a variety of purposes like indexing, locking, or recovery management. Consider, for example, a shutdown-management system which allows objects to register for notification when the system shuts down. To accomplish this, each participating object is registered with the manager, and the manager keeps a set of object identifiers that need to be notified. In the Bridge pattern example in figure 5.3 this could arise when the IconWindow class is registered and the corresponding XWindowImp class is also registered. If not special care is taken the XWindowImp class might get the shutdown call sent to it twice: once passed on from IconWindow and once directly from the shutdown manager.

Object schizophrenia is not necessarily a problem in itself but it can lead to several problems, if not special care is taken when using such objects. These problems can also be avoided by using multiple inheritance instead of the n-Bridge pattern when programming with facets.

5.3 Implementation with Multiple Inheritance

The UML example diagrams of facet-oriented designs in earlier chapters have been modeled with multiple inheritance. The product-classes inherits from one class in every facet. As noted above this can be directly implemented in a programming language that supports multiple inheritance. When using this method we lose the dynamic benefits of the n-Bridge solution but instead gain more control and avoid the object schizophrenia problem. In the window application example above it is obvious that the n-Bridge pattern is better suited for implementation than multiple inheritance, because all combinations of classes from the different hierarchies are usefull. We want to be able to use all window types on all plattforms. But if we want to control what combinations that can be used, multiple inheritance might be better, especially if we only want to allow a small subset of the possible combinations. The animal model in section 4.1, is one example when inheritance can be better, because the chosen class from each facet is unlikely to change. The dog will not suddenly learn to fly, so the need for dynamic exchangability is limited.

When using multiple inheritance to implement the faceted-model it would be nice if the compiler could help us to fullfill the demands of the model:

1. A facet-class can not inherit from a facet-class in another facet.
2. A product-class must inherit from exatly one class in every facet.
3. A product-class can not inherit from another product-class.
4. A facet-class can not inherit from a product class.

5.4 Language Support for Facets

In this section we look at what benefits can be gained from having support for facets in a programming language.

5.4.1 Pike

As an example of what kind of support a language can have for facets the Pike language has been extended to support facets. Pike is a dynamic language with a syntax similar to Java and C that supports multiple inheritance. Further information about the Pike language and the Pike compiler can be found at the Pike homepage: `http://pike.ida.liu.se`.

The standard syntax for defining two classes with inheritance in Pike is:

```
class A
{
// Body of class A
}
```

```
class B
{
inherit A;
// Body of class B
}
```

We now want to be able to specify that a class belongs to a given facet. We also want the compiler to check that if a class not belonging to any facet inherits from a class in a facet, it has to inherit from one class in all facets. Additionally, we want the compiler to check that a class in a facet does not inherit from a class in some other facet. In a large system it is possible that we want to use facets more than once, so we have to have some way of grouping facets together. For this purpose we use a *facet-group*. The syntax to specify that a class belongs to a given facet in a specific facet-group is:

```
class <class−name>
{
facet <facet−name> : <facet−group> ;
// Body of class
}
```

A class that inherits from a facet-class and does not contain the facet keyword as above is concidered to be a product class and must thus inherit from all facets. With this additional keyword the compiler can help the programer to keep the inheritance hierarchy correct when using facets.

Suppose we want to program the Bridge pattern, as shown in figure 5.3, with the added support for facets in Pike. From the figure we can see that we will need two facets: a Window facet and a WindowImp facet, belonging to the same facetgroup. Each facet will contain three classes and we will not have any product-classes as of yet. To avoid confusion we will call the Window facet WindowFacet and the WindowsImp facet WindowsImpFacet.

```
// The Window facet
class Window
{
  facet WindowFacet : BridgeFacetGroup;

  void DrawText()
  {
  // function body
  }
  void DrawRect()
  {
  // function body
  }
}

class IconWindow
{
  facet WindowFacet : BridgeFacetGroup;
  inherit Window;

  void DrawBorder()
  {
    DrawRect();
    DrawText();
  }
}

class TransientWindow
{
  facet WindowFacet : BridgeFacetGroup;
  inherit Window;

  void DrawBorder()
  {
    DrawRect();
  }
}

// The WindowImpFacet
class WindowImp
{
  facet WindowImpFacet : BridgeFacetGroup;
  void DevDrawText() { }
  void DevDrawRect() { }
}

class XWindowImp
{
  facet WindowImpFacet : BridgeFacetGroup;
  inherit WindowImp;

  void DevDrawText()
  {
    // function body
  }
  void DevDrawRect()
  {
    // function body
  }
}

class PMWindowImp
{
  facet WindowImpFacet : BridgeFacetGroup;
  inherit WindowImp;

  void DevDrawText()
  {
    // function body
  }
  void DevDrawRect()
  {
    // function body
  }
}
```

We now have to create product classes for all the different combinations of facet classes that our appliction will need. The PM-Window transient window class will for example have the following structure.

```
class PMTransientWindow
{
  inherit TransientWindow;
  inherit PMWindowImp;
  // Body of class
}
```

If we were to forget to inherit from a class in the WindowImpFacet facet the compiler would complain and tell us what was wrong with our code. Read more about the actual changes made to the Pike compiler to support facets in chapter 6.

Chapter 6

Extending Pike to Support Facets

In this chapter we will look in detail at the changes made to the Pike compiler to add support for facets as explained in section 5.4.1.

6.1 Introduction

To recapitulate, this is what is meant with a facet-class and a product-class:

facet-class: A class that contains the `facet` statement.

product-class: A class that inherits directly or indirectly from a facet-class. A class inheriting from a product-class is also a product-class.

To extend the Pike compiler to support facets the compiler will have to:

1. Parse the `facet` keyword and the tokens following on the same line.
2. Process the `facet` statement, i.e. remember that the class is a facet-class by adding the class name to some list of facet-classes.

3. Check if an inherit statement involves a facet-class.
    (a) If it does, check that the inherit is valid in regards to the facet model.
        i. If the inherit follows the facet model, do a normal inherit.
        ii. If the inherit violates the facet model, print an error message, and signal that the compilation failed.
    (b) If it does not, do a normal inherit.

Figure 6.1 shows the normal workflow of a compiler to the left, and to the right the modifications that were made at each step.

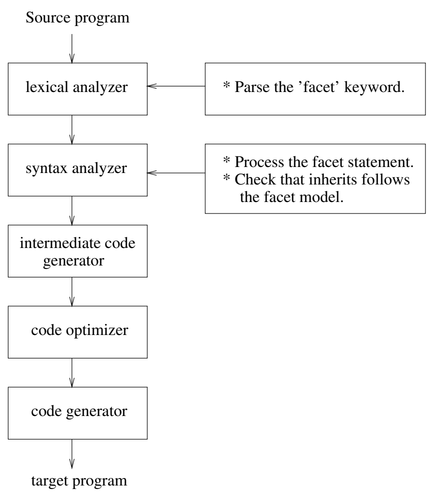

Source program
lexical analyzer
* Parse the 'facet' keyword.
syntax analyzer
* Process the facet statement.
* Check that inherits follows the facet model.
intermediate code generator
code optimizer
code generator
target program

Figure 6.1: Compiler flow chart.

The modifications done to the compiler does not affect the final code in any way.

6.2 Implementation

Every facet- and product-class stores whether it is a facet- or product-class and what facet-group it belongs to, facet-classes in addition also stores an identifier for the facet it belongs to. This information is stored in the program structure, defined in the file `program.h`.

```
struct program
{
  // ...
  INT16 facet_class;    /* PROGRAM_IS_X_CLASS (X=FACET/PRODUCT) */
  /* Identifier for the facet this facet class belongs to */
  INT32 facet_index;
  /* Pointer to the facet-group this class belongs to */
  struct object *facet_group;
};
```

In addition to using the facet statement we have to create a file called `<facet-group>.pmod`, for every facet-group we want to use. Each facet has to belong to a facet-group and facets that are to be used together must be placed in the same facet-group. Accordingly, facets that do not belong together have to belong to different facet-groups. The `<facet-group>.pmod` file must have the following content.

```
inherit facet_group;
int main() { }
```

This file is needed for the compiler to know what facet-groups there are in the system. An alternative would have been to let the compiler create this group automatically when it sees a facet-group name it does not know about. But to keep the changes to the compiler simple this method was chosen. This also avoids confusion if the programmer misspells the facet-group name in a facet statement. The compiler will not believe that this is the name of a new facet-group but complain that no such facet-group exists.

The facet_group class inherited from in the first statement of the facet-group file contains all necessary functionality to handle the facets, facet-classes and product-classes in that facet-group. This facet−group class is implemented as a Pike c-module. It contains data structures to store facet-classes and product-classes as well as functions to add a new facet-class, to add a new product-class and to check that all product-classes in the facet-group inherits from one facet-class in every facet. To allow this functionality two linked lists are kept in the module. One of known facets and their facet-classes and one of known product-classes and which facet-classes they inherits from.

6.2.1 Limitations

A limitation of the current solution is that it only allows a class to belong to one facet-group, because the classes themselves stores which facet-group they belong to. The reason for using this scheme is that we directly from the program structure can access the program's facet-group and thus easily call functions in the facet-group. For example when a non facet-class inherits from a facet-class we can access the facet-class' facet-group and call the function to add a product-class to the group. This limitation means that:

1. A class is not allowed to be a facet-class in one facet-group and a product-class in another.
2. A class can not be a facet-class in more than one facet-groups.
3. A class can not be a product-class in more than one facet-group.

Limitations one and three in this list could probably be useful to the programmer. For example, imagine if we want to model a computer using facets. We could then for example use the facets cpu, memory, input-output devices and network. To model the network we could use the model shown in section 4.2. This would mean that the product-classes in the network model at the same time would be facet-classes in the computer model. The facet-classes in the network model could be kept outside the network facet in the computer model. So we would get a model that looked something like in figure 6.2.

To support this, i.e. classes being product-classes and facet-classes at the same time, we would have to change the variable that stores which facet-group a class belongs to, in the program structure, into a linked list of facet-groups. To make the change simple we could have as standard that the first item in this list always was the group in which the class acted as facet-class, it would then be easy to know which group to use for what. The

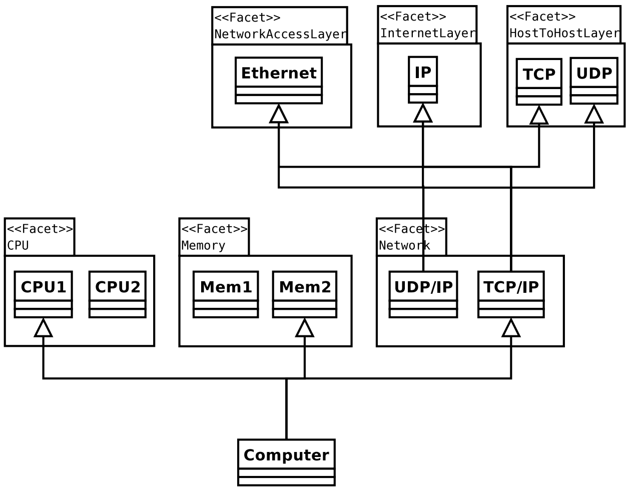


Figure 6.2: Computer model with several facet-groups.

variable that stores if this class is a facet- or product-class could remain the same, but with slightly different meaning. If it is set to facet-class we know that the first item in the facet-group list is its facet-class facet-group and the rest are facet-groups in which it acts as a product-class. If it is set to product-class we know that all the items in the facet-group list are facet-groups where this class acts as product-class. Furthermore, we would have to allow a facet-class in one facet-group to inherit from a facet-class in another facet-group.

To allow a class to be a facet-class in several facet-groups (item two in the list above), on the other hand, is probably not as useful. As in the computer model example above the facet-classes from the network model can be kept outside the network facet. As we do not want a product-class in the computer model to inherit directly from one of the facets in the network model, but rather from one of the product-classes of that model. If there are some cases where we want to allow a class to act as a facet-class in several facet-groups we would also have to change the format of the facet statement, so that we can specify several facet-groups. The product-classes would also need some way to tell which of the facet-groups, of the facet-class it inherits from, it wants to use. For the compiler to be able to check that it inherits from all facets in that facet-group.

6.2.2 Earlier Ideas

The first idea on how to add support for facets to the Pike compiler was to just add two linked list to the compiler for every facet-group. The first list consisting of facet-classes represented as structures with the name of the facet-class and the name of the facet it belonged to. The second list consisting of product-classes represented as structures with the name of the product-class and a linked list of names of the facet-classes it inherited from. This was before the realization that we might want to use facets several times in our application and thus need facet-groups. The solution could of course have been extended to support facet-groups. By letting, as in the current implementation, every class store whether it is a facet-class or a product-class and what facet-group it belongs to. The facet-groups could then be stored in a separate linked list with pointers to two lists of product- and facet-classes.

This solution probably would have worked fine but the code to support facets would be deeply intertwined with the rest of the compiler. In the current solution we have managed to isolate most of the code in a separate module. We would also have tainted the compilers namespace with additional function and variable names, most of which now are hidden in the facet-group. Otherwise the current solution is very similar to this first idea. A benefit of the current solution is also that the facet-group is an ordinary Pike object, and thus easy to manipulate from the compiler. If we instead would have stored the group name in a linked list we would have had to implement some functions to search for names in this list and to insert new names. This is now handled by the compiler and its symbol table.

The syntax of the facet statement has also undergone some change during the implementation. The first idea, before the need for facet-groups was understood, was to add the facet keyword directly after the class declaration,

```
class  Water facet ByEnvironment
{   // Body of class
}
```

But when the facet-group also had to be specified the line would have become to long. Instead the keyword was added as a new statement, and made to look like the inherit statement.

6.2.3 Compiler Output

With the current modifications to the Pike-compiler we get errors when:

- A facet-class inherits from a facet-class in a different facet.
- A facet-class inherits from a product-class.
- A product-class inherits from some but not all facets.
- A product-class inherits from more than one class in the same facet.
- A product-class inherits from another product-class, i.e. when a product-class inherits indirectly from the same facet more than once.
- A product- or facet-class inherits from a class in a different facet-group.
- The facet-group does not exist.

All the above applies for facets and product-classes in the same facet-group, when not stated otherwise.

For the actual code added to the Pike compiler see appendix A.

Chapter 7

Example Applications

In this chapter we will look at two examples of Pike programs that uses facets.

7.1 Travel Game

We will develop the base classes to simulate vehicles in a simple game. The goal of the game is to transport a number of persons from point A to point B before time runs out. An environment is given with land and sea areas, which can be either in sun or shadow, the environment also contains some vehicles that can be used to transport the persons. All persons on point A will have to be transported to point B before time runs out to succeed.

The vehicles have different characteristics that defines them, such as top speed, acceleration, maximum number of passengers, fuel and what environment it can move in, e.g. land, water or air. The vehicles can use one of three different fuels: gas, electricity via sun panels, or manual using pedals or oars. The vehicles that are not manually driven depend on the availability of their respective fuel, the gas will run out and the sun can be shadowed by clouds or other obstacles. All vehicles need a driver so the player will have to move persons to the vehicles and not the other way around.

7.1.1 Design

From these requirements we can create three facets to describe the vehicles:

- Movement with the classes land, water, and air.
- Fuel with the classes gas, electricity , and manual.
- Size with the classes personal for 1 passenger, medium for 1-2 passengers, and large for 1-6 passengers.

We will now go through each facet and look at how they will affect the final vehicle.

Movement Based on the vehicle's type of movement its top speed will be affected. This will be done by using a top speed modifier, that will be multiplied with a theoretical top speed for the vehicle type. The movement type will also affect in what environment the vehicle can move, so we might have to change vehicles at some point of the journey.

Fuel The fuel the vehicle uses will affect the vehicle's top speed and how far it can move or if it can only move in areas where the sun is shining.

Size The vehicle's size will affect how many passengers it can carry and the number of passengers will affect the vehicle's top speed. The more passengers on the vehicle the slower it will move.

This leads us to the class diagram shown in figure 7.1, in the diagram we also show three possible vehicles using the facet-classes.

7.1.2 Code

The design presented in the previous section gives us to the following Pike code.

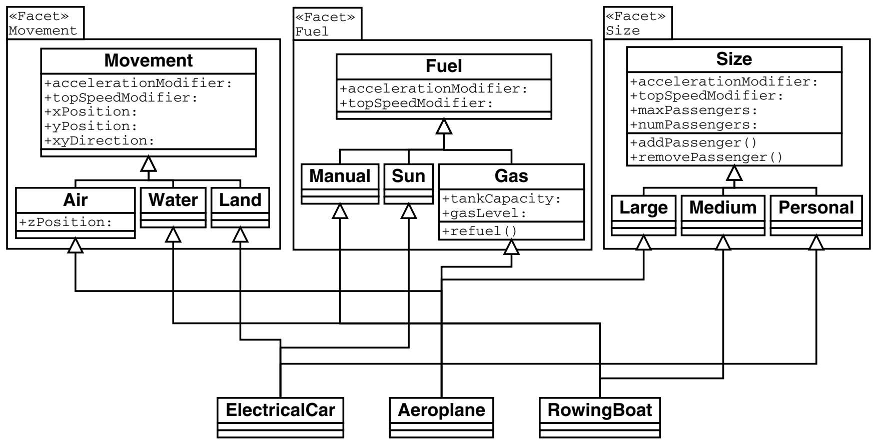


Figure 7.1: Facet-oriented design for vehicles.

Vehicle Facet

When we program with facets in Pike we have to create a facet group which the facets we want to use together have to belong to, as explaind in section 5.4. In this way we can have more than one set of facets in our application without the facets interfering with each other. The facet group is a standard Pike module that inherits from the system class facetgroup. To get the module to compile, it will also have to contain a main function.

```
inherit facetgroup;

int main() { return 0; }
```

To instruct the Pike compiler to look for the facet-group file in the same directory as the source files are in, we have to precede the facet-group name with a dot. So the facet statement will look like this:

```
class <name>
{
  facet <name> : .<group name>;
  // rest of class
}
```

If we want to get rid of the dot in front of the facet-group name we have to place the facet-group file somewhere in the Pike compilers search path. See the Pike language documentation for further details, `http://pike.ida.liu.se/docs`.

Fuel Facet

The Fuel facet contains four classes, a top-class Fuel that declares common attributes for classes in this facet and three sub-classes that models different types of fuel.

```
class Fuel
{
  facet Fuel : . VehicleFacets;
  float fuelSpeedModifier;
  static void create()
  {
    fuelSpeedModifier = 1.0;
  }
}

class Gas
{
  facet Fuel : . VehicleFacets;
  inherit Fuel;
  float tankCapacity;
  float gasLevel;
  float fuelConsumption;
  static void create()
  {
    :: create () /* The :: operator calls the function in all parent classes */
    tankCapacity = 150;
    gasLevel = tankCapacity;
    fuelConsumption = 1.0
  }
}

class Manual
{
  facet Fuel : . VehicleFacets;
  inherit Fuel;
  static void create()
  {
    fuelSpeedModifier = 0.1;
  }
}

class Sun
{
  facet Fuel : . VehicleFacets;
  inherit Fuel;
  static void create()
  {
    fuelSpeedModifier = 0.8;
  }
}
```

Movement Facet

The Movement facet contains four classes, a top-class Movement that specifies common attributes for classes in the facet and three sub-classes that defines different movement types.

```
class Movement
{
  facet Movement : .VehicleFacets;
  float movementSpeedModifier;
  flaot xPosition;
  float yPosition;
  float xyDirection;
  float currentSpeed;
  static void create()
  {
    movementSpeedModifier = 1.0;
    xPosition = 0;
    yPosition = 0;
    direction 0;
    currentSpeed = 0;
  }
}

class Air
{
  facet Movement : .VehicleFacets;
  inherit Movement;
  float zPosition;
  static void create()
  {
    movementSpeedModifier = 100.0;
    zPosition = 0;
  }
  int changeHeight(int toHeight)
  {
    if (toHeight != zPosition) {
      if (toHeight < 0) {
        write("The␣plane␣crashed\n");
        return −1;
      }
      else {
        zPosition = toHeight;
      }
    }
  }
}

class Land
{
  facet Movement : .VehicleFacets;
  inherit Movement;
  static void create()
  {
    movementSpeedModifier = 1.0;
  }
}

class Water
{
  facet Movement : .VehicleFacets;
  inherit Movement;
  static void create()
  {
    movementSpeedModifier = 0.1;
  }
}
```

Size Facet

The Size facet contains four classes, a top-class Size that specifies common attributes for classes in the facet and three sub-classes that defines different sizes.

```
class Size
{
  facet Size : . VehicleFacets;
  float sizeSpeedModifier;
  int numPassengers;
  int maxPassengers;
  static void create()
  {
    numPassengers = 0;
    maxPassengers = 0;
    setModifiers ();
  }
  void setModifiers()
  {
    sizeSpeedModifier = 1.0 − 0.2∗numPassengers;
  }
  int addPassenger()
  {
    if (numPassengers < maxPassengers) {
      numPassengers += 1;
      setModifiers ();
    }
    else
      write(”No␣room␣for␣more␣passengers␣in␣vehicle.\n”);
  }
  int removePassenger()
  {
    if (numPassengers > 0) {
      numPassengers −= 1;
      setModifiers ();
    }
    else
      write(”The␣vehicle␣is␣allready␣empty.\n”);
  }
}

class Personal
```

```
{
  facet Size  : . VehicleFacets;
  inherit Size;
  static void create()
  {
    numPassengers = 0;
    maxPassengers = 1;
    setModifiers ();
  }
}

class Medium
{
  facet Size  : . VehicleFacets;
  inherit Size;
  static void create()
  {
    numPassengers = 0;
    maxPassengers = 2;
    setModifiers ();
  }
}

class Large
{
  facet Size  : . VehicleFacets;
  inherit Size;
  static void create()
  {
    numPassengers = 0;
    maxPassengers = 6;
    setModifiers ();
  }
}
```

Product-Classes

The product-classes represents the actual vehicles in the game, by choosing one class from each facet the vehicles attributes are defined.

```
class RowingBoat
{
  inherit Manual;
  inherit Water;
  inherit Medium;
  private float topSpeed = 10.0;
  static void create()
  {
    :: create ();
  }
  float getTopSpeed()
  {
    return topSpeed *
      fuelSpeedModifier *
      movementSpeedModifier *
      sizeSpeedModifier;
  }
  int canMove(Environment env)
  {
    return numPassengers > 0 &&
      !env->onLand(xPosition, yPosition);
  }
}

class ElectricalCar
{
  inherit Sun;
  inherit Land;
  inherit Personal;
  private float topSpeed = 120.0;
  static void create()
  {
    :: create ();
  }

  float getTopSpeed()
  {
    return topSpeed *
      fuelSpeedModifier *
      movementSpeedModifier *
      sizeSpeedModifier;
  }
  int canMove(Environment env)
  {
    return env->sunShining(xPosition, yPosition) &&
      numPassengers > 0 &&
      env->onLand(xPosition, yPosition);
  }
}

class Aeroplane
{
  inherit Gas;
  inherit Air;
  inherit Medium;
  private float topSpeed = 1200.0;
  static void create()
  {
    :: create ();
  }

  float getTopSpeed()
  {
    return topSpeed *
      fuelSpeedModifier *
      movementSpeedModifier *
      sizeSpeedModifier;
  }
  int canMove(Environment env)
  {
    return gasLevel > 0 &&
      numPassengers > 0;
  }
  int moveTo(Environment env, float x, float y)
  {
    /* Check that the vehicle can move to x,y from the current
     * position , i.e. that x,y is on land and that we have enough fuel
     * to get there. */
     xPosition = x;
     yPosition = y;
  }
}
```

The Rest

To turn the above classes into an actual game we will need some more functionality.

- We will have to be able to keep track of who are in a certain vehicle.

- We will have to simulate the environment, where there is water and land, where the sun is shinging, where there are obstacles, etc.

- We will need some way to keep track of time, each action the user takes, such as moving people into or out of vehicles, could for example count as one clock tick.

The idea here was, however, not to develop a complete game but to show some actual Pike code that uses facets. Hopefully the above example has provided a better understanding of how facets can be used when programming in Pike.

7.2 A Review System for Research Papers

Physics, biology, and medicine all have well-refined public explanations of their research processes that provide guidance about what counts as "good research". But software engineering has not yet identified and explained either research processes or the way to recognize excellent work.

In an attempt to fill this gap Shaw, in her paper *What Makes good Research in Software* [18], suggests a way to classify computer science research papers based on the three areas: question, strategy/result, and validation. Question is the question that the paper tries to answer, such as "can X be done better?", "can X tell you Y?". Startegy/result is the result of the research, the answer to the question. This can be in the form of a new method, an analytic model, a tool, etc. Validation is how the result is validated in the paper. Is it with a thorough analysis, a report of some actual use of the result, or simply by persuaision?

As you might be able to see, this is a faceted model, allthough Shaw does not mention it in her paper. The three areas: question, strategy/result, and validation are the facets. In her paper Shaw suggests the following terms in the facets:

Question	Strategy/Result	Validation
Development method	Process	Analysis
Analysis method	Descriptive model	Experience
Evaluate instance	Analytic model	Example
Generalization	Empirical model	Evaluation
Feasibility	Tool	Persuasion
	Specific solution	
	Report	

Shaw further suggests that by looking at the classification of a paper with these facets we can say something about the quality of the paper. For example a common but bad plan for research papers is:

Question: can X be done better?, i.e. Development method.

Strategy/Result: new method, i.e. Process.

Validation: look, it works!, i.e. Persuasion.

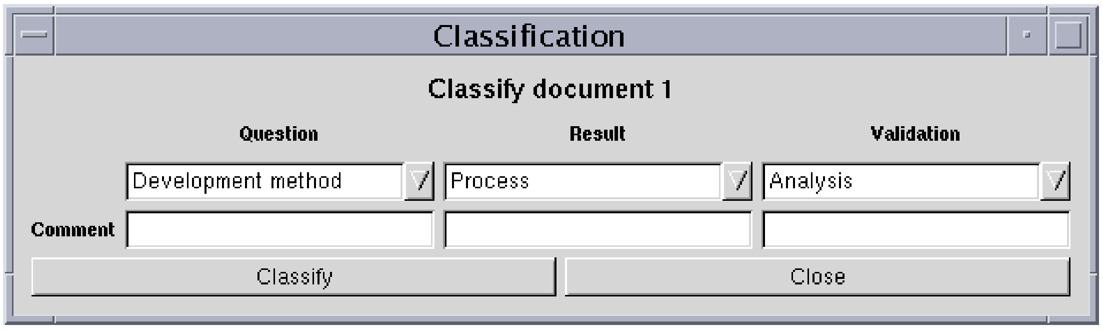
Persuasion is, however, not allways a bad validation method, we have to look at the combination of question, result and validation. The research strategies should be selected to match the research questions. If for example the paper wants to examine feasibility of a new method, persuasion might be a good way to show that the method actually works.

We will now describe a form-based GTK+ [1] application that lets a reviewer classify a research paper with the facets discussed above. This classification application should be seen as part of a bigger system for reviewing research papers.

Each term in the table above will be directly translated into a Pike facet-class. The paper will then be represented by a product-class. Our model will thus contain three facets, question, result, and validation, and 17 facet-classes. Because all combinations of the facet-classes from the different facets are legal, i.e. there is no restriction on how to select a class from

[1]The GIMP Toolkit `http://www.gtk.org`.

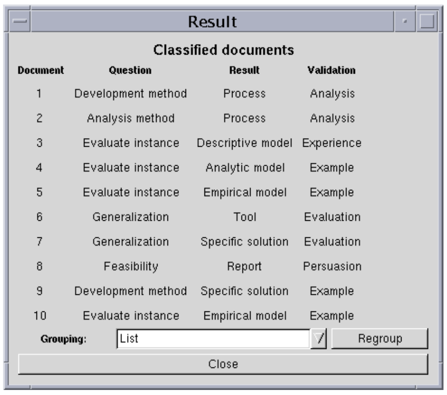
question, result and validation when creating a product class, we will get $5*7*5 = 175$ product-classes. Allthough each product-class will not contain any other code than **inherit** statements, it still would be pretty tiresome to manually write the code for all of these 175 classes. Therefore, a small program was written that takes a file with facet-classes as input and generates a file with all possible product-classes as output. The names of the product-classes generated is just the cross product of the names of the facet-classes it inherits from. So in this example we will get product-classes with names like FeasibilityToolAnalysis. This product-class will inherit from Feasibility in the Question facet, Tool in the Result facet and Analysis in the Validation facet. In addition to the product-classes the program also generates a mapping, a Pike data-structure, called string_to_class, that mapps from product-class names as strings to product-classes as programs. Using this mapping we can write statements like string_to_class [" FeasibilityToolAnalysis "]() to create an object of the FeasibilityToolAnalysis class. See appendix B for the complete code of the application.



Development method | Process | Analysis

Figure 7.2: Classification window.

Figure 7.2 shows the classification screen of the GTK+ application. Here the user gets to select a value for each of the facets from three drop down boxes, in addition the user can enter a comment. If we were to classify this masters thesis we could for example select Development method from the Question box and enter a comment that the method is about using facets in programming. In the current implementation of the application, documents are just given a number as identity and are not linked to any real documents or files on disk. After classifying 10 documents the result is shown, see figure 7.3.



Result

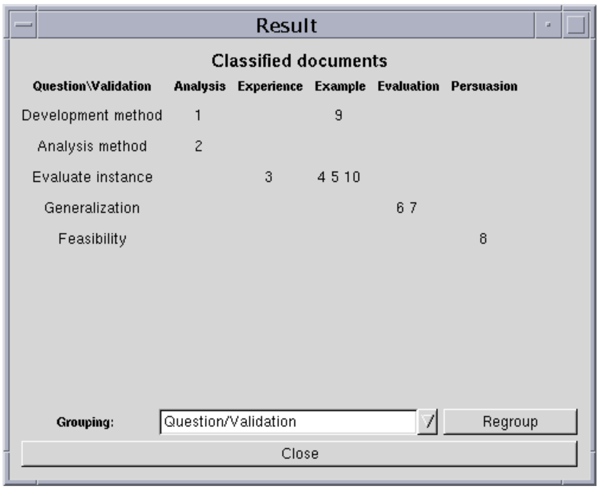
Classified documents

Document	Question	Result	Validation
1	Development method	Process	Analysis
2	Analysis method	Process	Analysis
3	Evaluate instance	Descriptive model	Experience
4	Evaluate instance	Analytic model	Example
5	Evaluate instance	Empirical model	Example
6	Generalization	Tool	Evaluation
7	Generalization	Specific solution	Evaluation
8	Feasibility	Report	Persuasion
9	Development method	Specific solution	Example
10	Evaluate instance	Empirical model	Example

Grouping: List

Figure 7.3: The result window.

From the drop-down list at the bottom of the window the user can select to view the result as List, Question/Result, Question/Validation or Result/Validation, see figure 7.4, where the result is grouped by Question and Validation. In the List view the user can also see the comments, as a tooltip, that was entered during the classification, by moving the pointer over any one of the fields.



Result

Classified documents

Question\Validation	Analysis	Experience	Example	Evaluation	Persuasion
Development method	1		9		
Analysis method	2				
Evaluate instance		3	4 5 10		
Generalization				6 7	
Feasibility					8

Grouping: Question/Validation

Figure 7.4: Result grouped by Question/Validation.

Instead of using facets and a product-class to represent each paper's classification we could simply have used an array for each classified paper, that stored the selected values from each facet. But the above solution was selected to demonstrate the use of facets in Pike.

Chapter 8

Conclusion

This chapter discusses the results from the thesis and look at what future work can be done.

8.1 Summary

The concept of separation of concerns is at the heart of software engineering. By identifying and encapsulating concerns in a system we hope to make it more extendible, reusable and easier to understand. In this thesis we drew inspiration from the field of information classification and the technique of faceted classification to come up with a somewhat new method to separate concerns in an object-oriented system. We showed how facets can be used in object-oriented design and how a facet-oriented design can be translated into program code. We saw further that the concept of a facet-oriented design is closely related to the Bridge, and n-Bridge design patterns. The differences from these patterns is that we use inheritance to access the functionality from the dimensions instead of aggregation. We finaly saw what kind of support a compiler can provide the programmer with when using facets in an object-oriented programming language with support for multiple inheritance.

8.2 Discussion

The purpose of this thesis was to examine how facets can be used in connection with object-oriented programming. From the summary above it should be clear how this has been done. And from the examples throughout the report we have been able to see various situations where a faceted model can be used in program design.

We have seen that some of the advantages of a faceted design includes:

- Extensibility, as the faceted model separates concerns into independent dimensions new functionality can be added by adding new classes to the facets, without having to recompile the other classes.
- By using facets in our design we are encouraged to think of our application in a way that helps to separate concerns, and to look for functionality that can benefit from beeing separated into different classes.
- By implementing our faceted design using inheritance we get good control over what combinations of classes from the facets that are legal to form.

8.3 Future Work

In this section we list some further ideas that has come up during the work on this thesis, that might be worth exploring.

1. Extend the support for facetes in Pike so that a class can belong to several facet-groups, see section 6.2.1.
2. Add support for partially instantiated product-classes, i.e. a class that inherits from some but not all facet-classes in a facet-group. This

is not supported in the current implementation. It could, however, be useful if several classes want to use a specific combiantion of classes from a subset of the facetes. The partiall product-class would then act as a kind of virtual class that we can not create instances of but only inherit from.

3. Further explore the usefullness of facets in program design and implementation by doing some larger examples with facets.

4. Add support for facets in another programming language.

5. In the TCP/IP example in section 4.2 we saw an example where all facet-classes contained the same function but with different contents, and the product class needed to call all of them in a predefined way. Perhaps the compiler automatically could create a similar function in the product class with the contents from the facet-classes' functions, or calls to them. In the TCP/IP example the product-classes would then get two functions: add_header() and remove_header() with the contents from, or calls to, the corresponding functions in the facet-classes in some specified order. If implemented this would be some sort of functional composition mechanism.

Appendix A

Pike Compiler Changes

In this appendix we look in more detail at the code that was added to the Pike compiler to support facets.

A.1 Program.h

The classes themselves stores if they are facet- or product-classes and if so which facet-group they belong to. This is stored in the struct program in the file `program.h`. With the current solution a class can only belong to one facet-group. This means that we can not create facets with facets in them, or let a class be both a product-class in one facet-group and a facet class in another facet-group, see section 6.2.1.

```
// ...
#define PROGRAM_IS_FACET_CLASS 0X1
#define PROGRAM_IS_PRODUCT_CLASS 0x2
// ...
struct program
{
  // ...
  INT16 facet_class;    /* PROGRAM_IS_X_CLASS (X=FACET/PRODUCT) */
  /* Index to the facet this facet class belongs to */
  INT32 facet_index;
  /* Pointer to the facet-group this class belongs to */
  struct object *facet_group;
};
```

facet_class is a flag that indicates if this class is a facet class or a product-class. facet_index is an index that identifies a facet in a facet-group. The facet_group pointer is used by product-classes so that we easily can add them to the correct facet-group.

A.2 Parsing

To support facets in the way described in section 5.4.1 the compiler needs to be able to parse the line:

facet <Facet name> : <Facet group> ;

Therefore the following code was added to `lexer.h`:

```
  // ...
 TWO_CHAR('f','a'):
    if(ISWORD("facet")) return TOK_FACET;
    break;
  // ...
```

This ensures that the compiler parses the facet keyword. To parse the rest of the line the following code was added to `language.yacc`:

```
// ...
%token TOK_FACET
// ...
%type <number> TOK_FACET
// ...
facet : TOK_FACET TOK_IDENTIFIER ':' idents ';'
{
  struct object *o;
  static int checked_product_classes = 0;
  if (Pike_compiler->compiler_pass == 1) {
    if (Pike_compiler->new_program->facet_class == PROGRAM_IS_FACET_CLASS) {
      yyerror("A class can only belong to one facet");
    }
    else {
      resolv_constant($4);
      if (Pike_sp[−1].type == T_OBJECT) {
        o = Pike_sp[−1].u.object;
        push_string($2−>u.sval.u.string);
        push_int(Pike_compiler−>new_program−>id);
        push_int(Pike_compiler−>new_program−>facet_class);
        apply(o, "add_facet_class", 3);
        if (Pike_sp[−1].type == T_INT &&
            Pike_sp[−1].u.integer >= 0) {
          Pike_compiler−>new_program−>facet_class = PROGRAM_IS_FACET_CLASS;
          Pike_compiler−>new_program−>facet_index = Pike_sp[−1].u.integer;
          Pike_compiler−>new_program−>facet_group = o;
        }
        else
          yyerror("Could␣not␣add␣facet␣class␣to␣system.");
        pop_stack();
      }
      else
        yyerror("Illegal␣facet␣group␣specifier.");
      free_node($4);
    }
  }
  else { // comiler_pass == 2
    if (!checked_product_classes) {
      apply(Pike_compiler−>new_program−>facet_group,
            "check_product_classes", 0);
      checked_product_classes = 1;
    }
  }
}
// ...
magic_identifiers3 :
// ...
  | TOK_FACET  { $$ = "facet"; }
// ...
bad_expr_ident:
// ...
  | TOK_FACET
  { yyerror("facet␣is␣a␣reserved␣word."); }
```

The first two lines adds the TOK_FACET symbol, that represents the facet keyword, to the system. The next line:

```
facet : TOK_FACET TOK_IDENTIFIER ':' idents ';'
```

specifies what is supposed to follow the facet keyword. The TOK_IDENTIFIER symbol matches the name of the facet and idents matches a Pike object. This object is the facet-group that the facet belongs to. This facet-group is a module that inherits from the class facetgroup, as explained in section 5.4.1. The last two sections ensures that the facet keyword is not used incorrectly.

A.3 The facetgroup Class

The facetgroup class is actually a Pike c module, but to Pike programmers it looks like an ordinary class that they can inherit from. The code for the module is in the file `facetgroup.cmod`. Besides code for managing facets and facet-classes it also contains code to manage product-classes.

```
/* -*- c -*-
|| This file is part of Pike. For copyright information see COPYRIGHT.
|| Pike is distributed under GPL, LGPL and MPL. See the file COPYING
|| for more information.
|| $Id: iterators.cmod,v 1.47 2003/09/05 15:19:20 mast Exp
*/

#include "global.h"
#include "fdlib.h"
#include "main.h"
#include "object.h"
#include "mapping.h"
#include "multiset.h"
#include "svalue.h"
#include "stralloc.h"
#include "array.h"
#include "pike_macros.h"
#include "pike_error.h"
#include "pike_memory.h"
#include "dynamic_buffer.h"
#include "interpret.h"
#include "las.h"
#include "gc.h"
#include "stralloc.h"
#include "security.h"
#include "opcodes.h"
#include "pike_error.h"
```

```
#include "program.h"
#include "operators.h"
#include "builtin_functions.h"
#include "constants.h"
#include "program.h"
#include "block_alloc.h"

DECLARATIONS

/*! @class facetgroup
 *!
 *! This class is used to handle facets in the system. All facets
 *! in the system have to belong to a facet group. The facet
 *! group is a Pike module that inherits from this class. For
 *! example you can create a file MyFacetgroup.pmod with the following
 *! content:
 *!
 *! @tt{inherit facetgroup;
 *! int main() { ; } @}
 *!
 *! You can then use the facet group MyFacetGroup in a facet-class
 *! like this:
 *!
 *! @tt{class A
 *! {
 *! facet NameOfMyFacet : .MyFacetGroup;
 *! // Rest of class A
 *! } @}
 *!
 */

/* Linked list of facet-classes */
struct facet_class_struct {
  int id;
  struct facet_class_struct *next;
};
/* Linked list of facets with their facet-classes */
struct facet_node_struct {
  struct pike_string *name;
  struct facet_class_struct *classes;
  struct facet_node_struct *next;
};
/* Linked list of facets a product-class is inheriting from */
struct facet_list_struct {
  int facet_index;
  int  facet_class ;
  struct  facet_list_struct  *next;
};
/* Linked list of product-classes with lists of what facets they use */
struct product_class_struct {
  int id;
  struct  facet_list_struct  *facets;
  int num_used_facets;
  struct product_class_struct *next;
};

BLOCK_ALLOC_FILL_PAGES(facet_class_struct, 2)
BLOCK_ALLOC_FILL_PAGES(facet_node_struct, 2)
BLOCK_ALLOC_FILL_PAGES(product_class_struct, 2)
BLOCK_ALLOC_FILL_PAGES(facet_list_struct, 2)


/* The actual facet_group class */
PIKECLASS facet_group
{
  CVAR struct facet_node_struct *facets;
  CVAR struct product_class_struct *pclasses;
  CVAR int num_facets;
  CVAR int checked_product_classes;

  /* Retruns 1 if product classes in this facet group have been checked,
   * 0 otherwise */
  PIKEFUN int product_classes_checked()
    {
      RETURN THIS->checked_product_classes;
    }
  /* Function to check that all product-classes inherits from all facet */
  PIKEFUN void check_product_classes()
    {
      struct  facet_list_struct  *fl;
      struct product_class_struct *pc;
      int error = 0;
      for(pc=THIS->pclasses; pc; pc = pc->next) {
        if (pc->num_used_facets < THIS->num_facets) {
          throw_error_object(low_clone(compile_callback_error_program),0,0,0,
                             "Product-class␣does␣not␣inherit␣from␣all␣facets.\n");
        }
        else if (pc->num_used_facets > THIS->num_facets) {
          /* It should be impossible to get here */
          throw_error_object(low_clone(compile_callback_error_program),0,0,0,
                              "Product−class␣inherits␣more␣than␣once␣from␣some␣facet.\n");
      }
    }
    THIS−>checked_product_classes = 1;
    pop_n_elems(args);
  }

/* Add info that the class "class" inherits from the facet "facet_index" */
PIKEFUN void add_product_class(int class, int facet_index, int facet_class)
  {
    struct facet_list_struct *fl, *fltmp;
    struct product_class_struct *pc, *pctmp;
    /* Set checked_product_classes to 0 to indicate that not all
     * product classes have been checked */
    THIS−>checked_product_classes = 0;
    /* Check whether the product−class is allready in our list of
     * product classes */
    for(pc=THIS−>pclasses; pc; pc = pc−>next) {
      if (class == pc−>id)
        break;
    }
    if (!pc) { /* New product−class */
      pctmp = alloc_product_class_struct();
      pctmp−>id = class;
      pctmp−>facets = NULL;
      pctmp−>num_used_facets = 0;
      pctmp−>next = THIS−>pclasses;
      THIS−>pclasses = pctmp;
      pc = pctmp;
    }
    /* pc now points to the product class to modify */
    for(fl=pc−>facets; fl; fl = fl−>next) {
      if (fl−>facet_index == facet_index)
        break;
    }
    if (!fl) {
      // Add facet "facet_index" to product−class "class"'s␣inherits
      fltmp = alloc_facet_list_struct ();
      fltmp−>facet_index = facet_index;
      fltmp−>facet_class = facet_class;
      pc−>num_used_facets++;
      fltmp−>next = pc−>facets;
      pc−>facets = fltmp;
    }
    else /* The product−class already inherits from this facet */
```

```
      if ( fl ->facet_class != facet_class )
        throw_error_object(low_clone(compile_callback_error_program ),0,0,0,
                            ”Product−class␣can␣only␣inherit␣from␣one␣class␣in␣every␣facet.\n”);
    pop_n_elems(args);
  }

/* Add a facet class in the facet group */
PIKEFUN int add_facet_class(string facet, int class , int flag )
  {
    struct facet_class_struct *c, *ctmp;
    struct facet_node_struct *f, *prevf, *ftmp;
    struct product_class_struct *pc, *prevpc;
    struct object *o;
    int facet_index = 0;
    if ( facet == NULL)
      throw_error_object(low_clone(compile_callback_error_program ),0,0,0,
                          ”Invalid␣facet␣name.\n”);

    // Check if it is a new facet or not.
    prevf = NULL;
    for(f=THIS->facets; f; f=f->next, facet_index++) {
      if (f->name == facet)
        break;
      prevf=f;
    }
    if ( flag == PROGRAM_IS_PRODUCT_CLASS) {
      // This occurs if the inherit statement comes before the facet
      // statement in the class
      for (prevpc=pc=THIS->pclasses; pc; pc = pc->next) {
        if (pc->id == class)
          break;
        prevpc = pc;
      }
      if (! pc)
        throw_error_object(low_clone(compile_callback_error_program ),0,0,0,
                            ”Program␣marked␣as␣product−class␣but␣not␣found␣in␣list␣of␣product−classe
      else {
        if (pc->num_used_facets > 1 ||
            facet_index != pc->facets->facet_index)
          throw_error_object(low_clone(compile_callback_error_program ),0,0,0,
                              ”Facet␣class␣can␣not␣inherit␣from␣a␣class␣in␣another␣facet.”);
        else {
          if (prevpc->id == pc->id)
            THIS->pclasses = pc->next;
          else
```

```
            prevpc->next = pc->next;
            really_free_product_class_struct (pc);
          }
        }
      }
      if (! f) { /* A new facet */
        THIS->num_facets++;
        f = alloc_facet_node_struct ();
        add_ref(f->name = facet);
        f->next = NULL;
        if (! prevf)
          THIS->facets = f;
        else
          prevf->next = f;
        f->classes = NULL;
      }
      // f now points to the facet 'facet' in 'facets'
      // Check whether the class 'class' is already in the facet
      for (c=f->classes; c; c = c->next) {
        if ( class == c->id)
          break;
      }
      if (c) {
        throw_error_object(low_clone(compile_callback_error_program),0,0,0,
                           "Redundant␣facet␣statement.\n");
      }
      else {
        ctmp = alloc_facet_class_struct ();
        ctmp->id = class;
        ctmp->next = f->classes;
        f->classes = ctmp;
      }
      RETURN facet_index;
    }

INIT
  {
    THIS->facets = NULL;
    THIS->pclasses = NULL;
    THIS->num_facets = 0;
    THIS->checked_product_classes = 0;
  }
EXIT
  {
    struct facet_node_struct *f, *fnext;
    struct facet_class_struct *fc, *fcnext;
    struct product_class_struct *p, *pnext;
    struct facet_list_struct *fl, *flnext;
    for (fnext=f=THIS->facets; fnext; f=fnext) {
      fnext = f->next;
      for (fcnext=fc=f->classes; fcnext; fc=fcnext) {
        fcnext = fc->next;
          really_free_facet_class_struct (fc);
      }
       really_free_string (f->name);
       really_free_facet_node_struct (f);
    }
    for (pnext=p=THIS->pclasses; pnext; p=pnext) {
      pnext = p->next;
      for (flnext=fl=p->facets; flnext; fl=flnext) {
        flnext = fl->next;
          really_free_facet_list_struct (fl);
      }
       really_free_product_class_struct (p);
    }
  }
};

void init_facetgroup(void)
{
    init_facet_class_struct_blocks ();
   init_facet_node_struct_blocks ();
    init_facet_list_struct_blocks ();
   init_product_class_struct_blocks ();
  INIT;
  add_global_program("facet_group", facet_group_program);
}

void exit_facetgroup(void)
{
  EXIT
}
/*! @endclass
 */
```

Normally you would use the function Pike_error() to signal an error in a Pike module. But Pike_error() throws a compilation_error error and this causes a lot of debug information to be printed. To avoid this we instead throw a compile_callback error directly.

A.4 Inheritance

The only additional code that is needed to make the compiler support facets is inheritance from facet- and product-classes. This is handled as part of the standard inheritance mechanism. After the ordinary consistensy checks for a standard inherit is done and before the program is actually inherited we check if we are inheriting from a facet class. This is done in the function check_for_facet_inherit () in the file `program.c`.

```
void check_for_facet_inherit (struct program *p)
{
  /* If the inherit statement comes before the facet keyword in the
   * class declaration the class will be temporarily marked as a
   * product-class. But this will be take care of when the facet
   * keyword is found. */
  if (p && p->facet_class == PROGRAM_IS_FACET_CLASS) {
    if (Pike_compiler->new_program->facet_class == PROGRAM_IS_FACET_CLASS) {
      if(Pike_compiler->new_program->facet_group == p->facet_group &&
         Pike_compiler->new_program->facet_index != p->facet_index)
        yyerror("Facet␣class␣can't␣inherit␣from␣class␣in␣different␣facet.");
    }
    /* Otherwise this is a product-class */
    else {
      int line = 0;
      Pike_compiler->new_program->facet_class = PROGRAM_IS_PRODUCT_CLASS;
      Pike_compiler->new_program->facet_group = p->facet_group;

      push_int(Pike_compiler->new_program->id);
      push_int(p->facet_index);
      push_int(p->id);
      apply(p->facet_group, "add_product_class", 3);
    }
  }
  else if (p && p->facet_class == PROGRAM_IS_PRODUCT_CLASS) {
    if (Pike_compiler->new_program->facet_class == PROGRAM_IS_FACET_CLASS &&
        Pike_compiler->new_program->facet_group == p->facet_group) {
      yyerror("Facet␣class␣can't␣inherit␣from␣product-class␣in␣same␣facet␣group.");
    }
    else if(Pike_compiler->new_program->facet_class==PROGRAM_IS_PRODUCT_CLASS){
      yyerror("Product-class␣can't␣inherit␣from␣other␣product-class.");
    }
  }
```

}

This function is called from low_do_inherit() in `program.h`.

Appendix B

The Paper Classification Program

In this appendix the code for the Pike-gtk application discussed in section 7.2 is shown along with the code for the program that was used to automatically generate the product-classes.

B.1 The Product-Class Generator

The program finds all facet-classes in a given file and generate all possible product-classes from them. The program also generates a mapping from product-class names as strings to the actual product classes. Facet-classes that contains the SKIP_FCLASS comment on the line above the **facet** statement will not be included in the generation.

```
int main()
{
  string SKIP_FCLASS = "//noautogenerate";
  write("Input␣filename␣where␣facet−classes␣can␣be␣found:␣");
  string file_name;
```

```
array(array(string)) facet_classes = ({});
mapping(string:int) facets = ([]);
int num_facets=0;
file_name = Stdio.stdin.gets();
string content;
if (content = Stdio.read_file(file_name)) {
  array(mixed) c_split, c_tmp;
  int pos;
  string cl, fac;
  // Parse input and remove " ", "\n" and "\t" from top level.
  c_split = filter(Parser.Pike.group(Parser.Pike.split(content),
                                     (["{":"}"])),
                   lambda(mixed t) { if (stringp(t)&&
                                         (t==" "||t=="\n"||t=="\t"))
                                     return 0;
                                     else return 1;});
  int i = 0;
  while ( (i = search( c_split , "class", i)) >= 0) {
    if (!stringp( c_split [i+1])) {
      write("Error in Pike program, no class name after class statement\n");
      exit (0);
    }
    write("Found class %s\n", c_split[i+1]);
    cl = c_split [i+1];
    if (!arrayp( c_split [i+2])) {
      write("Class without body: <%s>\n", c_split[i+2]);
      exit (0);
    }
    // remove " " from top level of the found class
    c_tmp = filter( c_split [i+2], lambda(mixed t) { if (stringp(t)&&(t==" "))
                                                   return 0;
                                                   else return 1;});
    // Is this a facet-class?
    if ( (( pos = search(c_tmp, "facet")) >= 0) &&
         !( stringp(c_tmp[pos-2]) &&
           (String.trim_all_whites(c_tmp[pos-2]) == SKIP_FCLASS)) ) {
      if (stringp(c_tmp[pos-2]))
        write("c_tmp pos-2: <" + c_tmp[pos-2] + ">\n");
      write("Found facet %s\n", c_tmp[pos+1]);
      fac = c_tmp[pos+1];
      // If this is a new facet, add it to the facets mapping
      if (zero_type( facets [fac])) {
        facets += ([fac:num_facets++]);
        facet_classes += ({({cl})});
      }
    else
      // Add the class to the facets-classes
      facet_classes [facets [fac]] += ({cl });
  }
  i+=3;
}
write("Parsed␣the␣file,␣generating␣product-classes...\n");
// Create the product-classes
if (num_facets < 2) {
  write("Less␣than␣two␣facets␣found,␣no␣product-classes␣created\n");
  exit (0);
}
int num_product_classes = 1;
for (int i = 0; i < num_facets; i++)
  num_product_classes *= sizeof(facet_classes[ i ]);
string pc_code = "";
array(string) product_classes, pc_tmp;
array(array(string)) pc_inherits;
/* Generate all legal combinations for the facet-classes found */
product_classes = sort( facet_classes [0]*
                        (num_product_classes/sizeof(facet_classes [0])));
pc_inherits = map(product_classes,
                  lambda(string s) { return ({s}); });
for (int i = 1; i < num_facets; i++) {
  pc_tmp = facet_classes[ i ] *
    (num_product_classes/sizeof(facet_classes [ i ]));
  product_classes = product_classes[*] + pc_tmp[*];
  pc_inherits = pc_inherits [*] +
    map(pc_tmp, lambda(string s) { return ({s}); })[*];
  product_classes = sort(product_classes , pc_inherits );
}
// begin the output with a mapping from the class name as a string
// to the class as program.
product_classes = ({"mapping(string:program)␣string_to_class␣=␣(["}) +
  product_classes;
// Add the mapping and the necessary statements around the
// product-class names
for (int i = 1; i <= num_product_classes; i++) {
  product_classes[0] += "\"" + product_classes[ i ] + "\":" + product_classes[ i ] + ",\n";
  product_classes [ i ] = "class␣" + product_classes[ i ] + "\n{\n";
  for (int j = 0; j < sizeof(pc_inherits [ i -1]); j++)
    product_classes [ i ] += "inherit␣" + pc_inherits [ i -1][j] + ";\n";
  product_classes [ i ] += "}\n";
}
/* Replace ",\n" with "]);\n" in the mapping */
```

```
    product_classes [0] = product_classes [0][0.. sizeof(product_classes[0])−3] + "]);\ n";
    write("File␣to␣write␣product_classes␣to:␣");
    file_name = Stdio.stdin.gets();
    Stdio.File f_out = Stdio.File();
    if (f_out−>open(file_name, "wc") < 0) {
      write("Unable␣to␣open␣file␣%s:␣%s\n", file_name, strerror(errno()));
      exit (0);
    }
    f_out−>write(product_classes);
    f_out−>close();
    write("Finished␣writing␣product−classes␣to␣file\n");
  }
  else {
    write("Unable␣to␣read␣file:␣<" + file_name + ">\n");
  }
}
```

B.2 The Classification Program

Here is the code for the classification program.

```
#include "classify_facets.pike"
int DOCS_TO_CLASSIFY = 10;
int RESULT_COLUMNS = 8;
array(Question) documents = ({});
array(string) questions = ({"Development␣method", "Analysis␣method",
                            "Evaluate␣instance", "Generalization",
                            " Feasibility "});
/* Mapping from question string to question facet−class name */
mapping(string:string) question_map = ([questions[0]:"Development",
                                        questions [1]: "QAnalysis",
                                        questions [2]: "QEvaluation",
                                        questions [3]: "Generalization",
                                        questions [4]: " Feasibility "]);
array(string) results = ({"Process", "Descriptive␣model", "Analytic␣model",
                          "Empirical␣model", "Tool", " Specific ␣solution",
                          "Report"});
/* Mapping from result string to result facet−class name */
mapping(string:string) result_map = ([results[0]:"Process",
                                      results [1]: "DescriptiveModel",
                                      results [2]: "AnalyticModel",
                                        results [3]: ”EmpiricalModel”,
                                        results [4]: ”Tool”,
                                        results [5]: ” SpecificSolution ”,
                                        results [6]: ”Report”]);
array(string) validations = ({”Analysis”, ”Experience”, ”Example”,
                                    ”Evaluation”, ”Persuasion”});
/* Mapping from validation string to validation facet−class name */
mapping(string:string) validation_map = ([validations[0]:”VAnalysis”,
                                           validations [1]: ”Experience”,
                                           validations [2]: ”Example”,
                                           validations [3]: ”VEvaluation”,
                                           validations [4]: ”Persuasion”]);

/* Main creates the classification window and sets up bindings from the
   buttons to functions */
int main()
{
  GTK.Window win;
  GTK.Button btn1, btn2;
  GTK.Box vbox1, hbox1, vb0, vb1, vb2, vb3, hbox2;
  GTK.Combo cmb1, cmb2, cmb3;
  GTK.Entry entr1, entr2, entr3;
  GTK.setup_gtk();
  win = GTK.Window(GTK.WINDOW_TOPLEVEL);
  win−>set_title(” Classification ”);
  win−>set_border_width(10);

  vbox1 = GTK.Vbox(0,5);
  hbox1 = GTK.Hbox(0,5);
  hbox2 = GTK.Hbox(0,5);
  GTK.parse_rc(”style␣\”title\”␣{␣fg[NORMAL]␣=␣{0.0,0.0,0.0}␣font␣=␣\”−adobe−helvetica−bold
  win−>add(vbox1);

  int doc_num = 1;
  GTK.Label Heading1 = GTK.Label(”Classify␣document␣1”)−>set_name(”Heading1”);
  vbox1−>pack_start_defaults(Heading1);
  vbox1−>pack_start_defaults(hbox1);
  vbox1−>pack_start(hbox2,0,1,0);
  vb0 = GTK.Vbox(1,5);
  vb1 = GTK.Vbox(1,5);
  vb2 = GTK.Vbox(1,5);
  vb3 = GTK.Vbox(1,5);
  hbox1−>pack_start_defaults(vb0);
  hbox1−>pack_start_defaults(vb1);
  hbox1−>pack_start_defaults(vb2);
```

```
hbox1->pack_start_defaults(vb3);
vb0->pack_start_defaults(GTK.Label(""));
vb0->pack_start_defaults(GTK.Label(""));
vb0->pack_start_defaults(GTK.Label("Comment")->set_name("tblHeading"));
vb1->pack_start_defaults(GTK.Label("Question")->set_name("tblHeading"));
cmb1 = GTK.Combo();
cmb1->set_value_in_list(1,0);
cmb1->set_popdown_strings(questions);
cmb1->entry()->set_editable(0);;
vb1->pack_start_defaults(cmb1);
entr1 = GTK.Entry();
vb1->pack_start_defaults(entr1);
vb2->pack_start_defaults(GTK.Label("Result")->set_name("tblHeading"));
cmb2 = GTK.Combo();
cmb2->set_value_in_list(1,0);
cmb2->set_popdown_strings(results);
cmb2->entry()->set_editable(0);
vb2->pack_start_defaults(cmb2);
entr2 = GTK.Entry();
vb2->pack_start_defaults(entr2);
vb3->pack_start_defaults(GTK.Label("Validation")->set_name("tblHeading"));
cmb3 = GTK.Combo();
cmb3->set_value_in_list(1,0);
cmb3->set_popdown_strings(validations);
cmb3->entry()->set_editable(0);
vb3->pack_start_defaults(cmb3);
entr3 = GTK.Entry();
vb3->pack_start_defaults(entr3);
btn1 = GTK.Button("Classify");
btn1->signal_connect("clicked", classify ,
                         ({ ({cmb1->entry(), entr1}),
                               ({cmb2->entry(), entr2}),
                               ({cmb3->entry(), entr3}) }) );


function change_heading =
  lambda() {
    if (doc_num < DOCS_TO_CLASSIFY) {
      array(mixed) ch = win->child()->children();
      ch[0]->set_text("Classify␣document␣" + ++doc_num);
    }
    else {
      win->destroy();
      show_result();
    }
    };
  btn1−>signal_connect("clicked", change_heading);
  hbox2−>pack_start_defaults(btn1);
  btn2 = GTK.Button("Close");
  btn2−>signal_connect("clicked", lambda() { win−>destroy(); });
  hbox2−>pack_start_defaults(btn2);
  win−>show_all();
  return −1;
}

/∗ This function takes an array with three names of facet−classes
 ∗ and adds the corresponding product−class to the documents array ∗/
void classify (array(array(mixed)) e)
{
  if (!objectp(e [0][0])) {
    write("Unable␣to␣read␣choices␣from␣combo␣boxes.\n");
    return;
  }
  string dclass_name = question_map[e[0][0]−>get_text()] +
    result_map[e[1][0]−>get_text()] +
    validation_map[e[2][0]−>get_text ()];
  Question d = string_to_class [dclass_name]();
  d−>qcomment = e[0][1]−>get_text();
  d−>rcomment = e[1][1]−>get_text();
  d−>vcomment = e[2][1]−>get_text();
  e[0][1]−>set_text ("");
  e[1][1]−>set_text ("");
  e[2][1]−>set_text ("");
  documents += ({ d });
}

/∗ This function creates the result window ∗/
int show_result()
{
  GTK.Window win;
  GTK.Button btn, btn2;
  GTK.Table tbl;
  GTK.Box vbox, hbox;
  GTK.Tooltips tt;
  GTK.Label q,r,v;
  GTK.Combo cb;
  array(string) grouping = ({"List", "Question/Result", "Question/Validation", "Result/Validation"}
  win = GTK.Window(GTK.WINDOW_TOPLEVEL);
  win−>set_title("Result");
  win−>set_border_width(10);
  tbl = GTK.Table(RESULT_COLUMNS, DOCS_TO_CLASSIFY+1, 0);
  /* Initialize table with labels and event boxes */
  for (int i = 0; i < RESULT_COLUMNS*(DOCS_TO_CLASSIFY+1); i++) {
    if (i < RESULT_COLUMNS) // The first row
      tbl->attach_defaults(GTK.Label("")->set_name("tblHeading"),i,i+1,0,1);
    else
      tbl->attach_defaults(GTK.EventBox()->add(GTK.Label("")),
                           i%RESULT_COLUMNS,
                           (i%RESULT_COLUMNS)+1,
                           i/RESULT_COLUMNS,
                           (i/RESULT_COLUMNS)+1);
  }
  vbox = GTK.Vbox(0,5);
  win->add(vbox);
  vbox->pack_start_defaults(GTK.Label("Classified documents")->set_name("Heading"));
  tt = GTK.Tooltips();
  group_result(({GTK.Entry()->append_text(grouping[0]), tbl, tt}));
  tbl->set_col_spacings(10);
  tbl->set_row_spacings(10);
  vbox->pack_start_defaults(tbl);
  hbox = GTK.Hbox(0,5);
  hbox->pack_start_defaults(GTK.Label("Grouping: ")->set_name("tblHeading"));
  cb = GTK.Combo()->set_value_in_list(1,0)->set_popdown_strings(grouping);
  cb->entry()->set_editable(0);
  hbox->pack_start_defaults(cb);
  btn2 = GTK.Button("Regroup");
  btn2->signal_connect("clicked", group_result, ({cb->entry(), tbl, tt}));
  hbox->pack_start_defaults(btn2);
  vbox->pack_start_defaults(hbox);
  btn = GTK.Button("Close");
  btn->signal_connect("clicked", lambda() { win->destroy(); exit(0); });
  vbox->pack_start(btn,0,0,0);
  win->signal_connect("destroy", exit, 0);
  win->show_all();
  return -1;
}

/* This function re-groups the result in the result window according to
   the users selection */
void group_result(array(mixed) args)
{
  array(string) questions = ({});
  array(string) results = ({});
  array(string) validations = ({});
  int i,j,k = 0;
```

```
GTK.EventBox qe, re, ve;
string new_gr = args[0]−>get_text();
GTK.Table tbl = args[1];
GTK.Tooltips tt = args[2];
array(mixed) ch = reverse(tbl−>children());
int index;
string cell = ””;
array(array(array(int))) tbl_docs = ({ });
int r,c;
clear_table (tbl );
switch(new_gr)
  {
  case ”List”:
    tt−>enable();
    ch[0]−>set_text(”Document”);
    ch[1]−>set_text(”Question”);
    ch[2]−>set_text(”Result”);
    ch[3]−>set_text(”Validation”);
    for(i = 0; i < sizeof(documents); i++) {
      index = RESULT_COLUMNS∗(i+1);
      ch[index]−>child()−>set_text(i+1 + ”␣”);
      ch[index+1]−>child()−>set_text(documents[i]−>question);
      ch[index+2]−>child()−>set_text(documents[i]−>result);
      ch[index+3]−>child()−>set_text(documents[i]−>validation);
      if (documents[i]−>qcomment != ””)
        tt−>set_tip(ch[index+1], documents[i]−>qcomment);
      if (documents[i]−>rcomment != ””)
        tt−>set_tip(ch[index+2], documents[i]−>rcomment);
      if (documents[i]−>vcomment != ””)
        tt−>set_tip(ch[index+3], documents[i]−>vcomment);
    }
    break;
  case ”Question/Result”:
    tt−>disable();
    for (i = 0; i < sizeof(documents); i++) {
      if ( (r = search(questions, documents[i]−>question)) < 0) {
        questions += ({documents[i]−>question});
        r = sizeof(questions)−1;
      }
      if ( (c = search( results , documents[i]−>result)) < 0) {
        results += ({documents[i]−>result});
        c = sizeof( results )−1;
      }
      if (sizeof(tbl_docs) <= r) {
        tbl_docs += ({({({i+1})})});
        for (j = 0; j < c; j++)
          tbl_docs[r] = ({({})}) + tbl_docs[r];
      }
      else if (sizeof(tbl_docs[r]) <= c) {
        for (j = sizeof(tbl_docs[r]); j < c; j++)
          tbl_docs[r] += ({({})});
        tbl_docs[r] += ({({i+1})});
      }
      else {
          tbl_docs[r][c] += ({i+1});
      }
    }
    ch[0]->set_text("Question\\Result");
    /* First the table headings */
    for (i = 0; i < sizeof(results); i++) {
      ch[i+1]->set_text(results[i]);
    }
    /* Then all the cells */
    for (i = 0; i < sizeof(questions); i++) {
      index = RESULT_COLUMNS*(i+1);
      /* The questions side heading first on every row */
      ch[index]->child()->set_text(questions[i]);
      for (j = 0; j < sizeof(tbl_docs[i]); j++) {
        cell = "";
        for (k = 0; k < sizeof(tbl_docs[i][j]); k++)
          cell += tbl_docs[i][j][k] + "␣";
        ch[index+j+1]->child()->set_text(cell);
      }
    }
    break;
  case "Question/Validation":
    tt->disable();
    for (i = 0; i < sizeof(documents); i++) {
      if ( (r = search(questions, documents[i]->question)) < 0) {
        questions += ({documents[i]->question});
        r = sizeof(questions)-1;
      }
      if ( (c = search(validations, documents[i]->validation)) < 0) {
        validations += ({documents[i]->validation});
        c = sizeof(validations)-1;
      }
      if (sizeof(tbl_docs) <= r) {
        tbl_docs += ({({({i+1})})});
        for (j = 0; j < c; j++)
          tbl_docs[r] = ({({})}) + tbl_docs[r];
        }
        else if (sizeof(tbl_docs[r]) <= c) {
          for (j = sizeof(tbl_docs[r]); j < c; j++)
            tbl_docs[r] += ({({})});
          tbl_docs[r] += ({({i+1})});
        }
        else {
          tbl_docs[r][c] += ({i+1});
        }
      }
      ch[0]->set_text("Question\\Validation");
      /* First the table headings */
      for (i = 0; i < sizeof(validations); i++) {
        ch[i+1]->set_text(validations[i]);
      }
      /* Then all the cells */
      for (i = 0; i < sizeof(questions); i++) {
        index = RESULT_COLUMNS*(i+1);
        /* The questions side heading first on every row */
        ch[index]->child()->set_text(questions[i]);
        for (j = 0; j < sizeof(tbl_docs[i]); j++) {
          cell = "";
          for (k = 0; k < sizeof(tbl_docs[i][j]); k++)
            cell += tbl_docs[i][j][k] + "␣";
          ch[index+j+1]->child()->set_text(cell);
        }
      }
      break;
    case "Result/Validation":
      tt->disable();
      for (i = 0; i < sizeof(documents); i++) {
        if ( (r = search(results, documents[i]->result)) < 0) {
          results += ({documents[i]->result});
          r = sizeof(results)-1;
        }
        if ( (c = search(validations, documents[i]->validation)) < 0) {
          validations += ({documents[i]->validation});
          c = sizeof(validations)-1;
        }
        if (sizeof(tbl_docs) <= r) {
          tbl_docs += ({({({i+1})})});
          for (j = 0; j < c; j++)
            tbl_docs[r] = ({({})}) + tbl_docs[r];
        }
        else if (sizeof(tbl_docs[r]) <= c) {
          for (j = sizeof(tbl_docs[r]); j < c; j++)
            tbl_docs[r] += ({({})});
          tbl_docs[r] += ({({i+1})});
        }
        else {
            tbl_docs[r][c] += ({i+1});
        }
      }
      ch[0]->set_text("Result\\Validation");
      /* First the table headings */
      for (i = 0; i < sizeof(validations); i++) {
        ch[i+1]->set_text(validations[i]);
      }
      /* Then all the cells */
      for (i = 0; i < sizeof(results); i++) {
        index = RESULT_COLUMNS*(i+1);
        ch[index]->child()->set_text(results[i]);
        for (j = 0; j < sizeof(tbl_docs[i]); j++) {
          cell = "";
          for (k = 0; k < sizeof(tbl_docs[i][j]); k++)
            cell += tbl_docs[i][j][k] + "␣";
          ch[index+j+1]->child()->set_text(cell);
        }
      }
      break;
    }
  tbl->show_all();
}

/* this function clears the result window result table */
void clear_table(GTK.Table tbl)
{
  array(mixed) ch = tbl->children();
  GTK.EventBox eb = GTK.EventBox();
  GTK.Label l = GTK.Label("");
  for (int i = 0; i < sizeof(ch); i++) {
    /* Is the child an event box? */
    if (object_program(ch[i])==object_program(eb))
      ch[i]->child()->set_text("");
    /* Otherwise it is just a label */
    else
      ch[i]->set_text("");
  }
}
```

Here is the facet- and product-classes, only one generated product class is shown but with some imagination the reader should be able to guess the structure of the left out ones. When this file was used with the auto-generation program above, SKIP_FCLASS comments were added in the Question, Result and Validation classes.

```
class Question { facet Question:.MyFacetGroup; string qcomment; }
class Development { facet Question:.MyFacetGroup; inherit Question;
  string question = "Development␣method"; }
class QAnalysis { facet Question:.MyFacetGroup; inherit Question;
  string question = "Analysis␣method"; }
class QEvaluation { facet Question:.MyFacetGroup; inherit Question;
  string question = "Evaluate␣instance"; }
class Generalization { facet Question:.MyFacetGroup; inherit Question;
  string question = "Generalization"; }
class Feasibility { facet Question:.MyFacetGroup; inherit Question;
  string question = "Feasibility"; }
class Result { facet Result:.MyFacetGroup; string rcomment; }
class Process { facet Result:.MyFacetGroup; inherit Result; string
  result = "Process"; }
class DescriptiveModel { facet Result:.MyFacetGroup; inherit Result;
  string result = "Descriptive␣model"; }
class AnalyticModel { facet Result:.MyFacetGroup; inherit Result;
  string result = "Analytic␣model"; }
class EmpiricalModel { facet Result:.MyFacetGroup; inherit Result;
  string result = "Empirical␣model"; }
class Tool { facet Result:.MyFacetGroup; inherit Result; string result
  = "Tool"; }
class SpecificSolution { facet Result:.MyFacetGroup; inherit Result;
  string result = "Specific␣solution"; }
class Report { facet Result:.MyFacetGroup; inherit Result; string
  result = "Report"; }
class Validation { facet Validation:.MyFacetGroup; string vcomment; }
class VAnalysis { facet Validation:.MyFacetGroup; inherit Validation;
  string validation = "Analysis"; }
class Experience { facet Validation:.MyFacetGroup; inherit Validation;
  string validation = "Experience"; }
class Example { facet Validation:.MyFacetGroup; inherit Validation;
  string validation = "Example"; }
class VEvaluation { facet Validation:.MyFacetGroup; inherit
  Validation; string validation = "Evaluation"; }
class Persuasion { facet Validation:.MyFacetGroup; inherit Validation;
  string validation = "Persuasion"; }
```

```
/* Automatically generated product-classes with names as cross products
 * of their ancestors, in the order: Question*Result*Validation */

mapping(string:program) string_to_class = ([
"DevelopmentAnalyticModelExample":DevelopmentAnalyticModelExample
// Rest of mapping not shown.
]);
class DevelopmentAnalyticModelExample
{
inherit Development;
inherit AnalyticModel;
inherit Example;
}
// 174 product-classes not shown.
```

Bibliography

[1] S. Araban, A. S. M. Sajeev, and J. Chen. Tool support for class library reuse. In *Technology of Object-Oriented Languages and Systems*, pages 159–170, 1995.

[2] L. Bermans and M. Aksit. Composing multiple concerns using composition filters. *Communications of the ACM*, 44(10):51–57, Oct. 2001.

[3] M. Dewey. *Decimal classification and relative index.* Forest Press, 22 edition, 2003.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[5] L. Guttman. An outline of some new methodology for social research. *Public Opinion Quaterly*, 18(4):395–404, 1954.

[6] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). *Proceeding of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 411–428, Sept. 1993. ACM.

[7] Hyperspace programming web site. `http://www.research.ibm.com/hyperspace`, May 2004.

[8] G. Kiczales. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, 1997. Invited presentation.

[9] A. L. Opdahl. Multi-perspective modelling of requirements: A case study using facet models. In *The 3rd Australian Conference on Requirements Engineering*, 1998.

[10] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. Research Report 21452, IBM, Apr. 1993.

[11] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 12(15):1053–1058, Dec. 1972.

[12] R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, May 1991.

[13] U. Priss. Faceted knowledge representation. *Electronic Transactions on Artificial Intelligence*, 2000(4):21–33, 2000.

[14] S. R. Ranganathan. *Elements of library classification.* Asia Publishing House, 1962.

[15] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with objects: the OOram software engineering method.* Prentice Hall, 1996.

[16] D. Riehle and T. Gross. Role model based framework design and integration. In *Object-Oriented Programming, systems, Languages, and Applications*, 1998.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design.* Prentice Hall, 1991.

[18] M. Shaw. What makes good research in software engineering? In *European Joint Conference on Theory and Practice of Software (ETAPS 2002)*, pages 1–7, 2002.

[19] Subject-oriented programming web site. `http://www.research.ibm.com/sop/sopcpats.htm`, May 2004.

LINKÖPINGS UNIVERSITET	**Avdelning, Institution** Division, Department IDA, Dept. of Computer and Information Science 581 83 Linköping	**Datum** Date May 17, 2004

Språk Language ☐ Svenska/Swedish ☒ Engelska/English ☐ ______	**Rapporttyp** Report category ☐ Licentiatavhandling ☒ Examensarbete ☐ C-uppsats ☐ D-uppsats ☐ Övrig rapport ☐ ______	**ISBN** — **ISRN** LITH-IDA-EX–04/047–SE **Serietitel och serienummer** Title of series, numbering **ISSN** —

URL för elektronisk version
`http://www.ep.liu.se/exjobb/ida/2004/dd-d/047/`

Titel	Facetorienterad design
Title	Facet-Oriented Program Design
Författare Author	Mikael Amborn

Sammanfattning
Abstract

This thesis examines the concept of separation of concerns in software engineering. By taking inspiration from the field of information classification and the technique of faceted classification it suggest a new method to separate concerns in software. The method, which we call facet-oriented programming, is described and various example of its use is shown. Possible support for facets in a programming language is discussed and the discussed features are then implemented in a compiler for the Pike programming language.

Nyckelord Keywords: software design, separation of concerns, faceted classification, Pike, multiple inheritance