SCALA & DESIGN PATTERNS

(EXPLORING LANGUAGE EXPRESSIVITY)

MASTERS THESIS

FREDRIK SKEEL LØKKE 20022555

ADVISOR: ERIK ERNST 30. MARTS 2009

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS

Abstract

This thesis explores a wellknown catalog of design patterns in the context of the programming language Scala. Design patterns do in general not exist as reusable components. Scala features novel abstraction mechanisms with a strong focus on the writing of components and is thus interesting to explore in the context of design patterns.

General implementation problems concerning design patterns are identified. The analysis of the catalog shows that Scala can deal with some of these problems. Components for three patterns are provided, but in general Scala lacks the needed abstraction mechanisms.

The principles behind design patterns, has long been regarded as sound engineering advice. The second principle, “Favor object composition over class inheritance”, is shown to be weakened in Scala. Further, a language feature proposal is presented, that if present in Scala in its full generality would result in the need for reevaluation of the principle.

Dansk Referat Dette speciale analyserer et velkendt katalog af design mønstre, i kontekst af programmeringssproget Scala. Design mønstre eksisterer generelt ikke som genanvendelige komponenter. Scala indeholder sprogkonstruktioner med et stærkt fokus påskrivningen af komponenter og er derfor interessant at udforske i design mønstre sammenhænge.

Generelle implementations problemer vedrørende design mønstre identificeres. Analysen af kataloget viser at Scala kan håndtere visse af problemerne. Komponenter for tre mønstre præsenteres, men generelt mangler Scala de nødvendige abstraktionsmekanismer.

Principperne bag design mønstre har længe været opfattet som sunde. Det andet princip, "Favoriser objekt komposition over klasse nedarvning", viser sig at være mindre betydningsfuld i Scala. Ydermere, et forslag til en ny sprogkonstruktion i Scala er præsenteret, der hvis gennemført ville resultere i nødvendigheden af en reevaluering af princippet.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis will explore *design patterns* in the context of the programming language *Scala*.

Motivation A key property of design patterns is that they in general do not exist as reusable components. They must be reimplemented in a specific context each time they are applied to a design. General problems have been identified with existing design pattern implementations, that relate to how easily the concepts involved are expressed in the language.

Scala [18] is a multiparadigm language that mixes object-orientation with functional features and introduces novel concepts. The new language constructs has a focus on providing abstractions for writing components [15] and are thus interesting to explore in the context of design patterns. *Pattern componentization*[3][p.11] refers to the process of turning a design pattern into a component.

Do the new constructs enable us to componentize patterns? Which leads us to the question of, when should a pattern be componentized in the first place? Futhermore, can we deal with the general implementation problems in Scala?

Design patterns center around object-oriented design, but some of Scalas features are only known from functional languages. Are these features usable and applicable in an object-oriented world? Do we gain anything from this unification of OO and functional concepts when applied to Design Patterns?

Central to design patterns are the principles behind them. Does the second principle "Favor object composition over class inheritance" [9][p. 20] apply to Scala as well? A wellknown catalog [9] of Design Patterns will be used as the basis for the exploration.

Purpose The thesis goals can be summarised as follow.

1. Identify known implementation problems with design patterns.

2. Give an overview of the novel features in Scala, in order to

3. Analyse a collection of Design Patterns in the context of Scala. For each pattern provide an improved implementation or possibly a componentized pattern.

4. With the questions from the motivation in mind, present and identify findings in the analysis as a whole.

5. Discuss which features not present in Scala would be useful for the componentization and/or improvement of design pattern implementations in general.

Prerequisites We assume that the reader is familiar with a mainstream object-oriented language such as Java or C++, and to a more limited degree with concepts from functional languages. Familarity with Design Patterns in general is assumed as well.

Structure The thesis consist of 3 parts.

Part I introduces Design Patterns. We identify known problems and proposes solutions to these problems. Secondly, Scala is introduced with a special focus on the novel constructs.

Part II starts by describing the analysis that will be performed on the Design Patterns catalog, and proceeds with the actual analysis of each pattern.

Finally, Part III presents related work. Further, we present findings in the analysis and try to answer the questions stated in the motivation. We discuss beneficial language features not present in Scale and we conclude the thesis.

Practical Scala is freely available and can be obtained from:
`http://www.scala-lang.org/`.
Installation instructions for an eclipse plugin [19] here:
`http://www.scala-lang.org/node/94`.

Part I

Design Patterns and Scala

Chapter 2

Design Patterns

This chapter will introduce and explain the rationale behind design patterns. The principles behind will be presented. The qualities, formal as well as informal, will be discussed. Several problems with design patterns will be identified and potential solutions to these problems will be shown.

2.1 Introduction

A design pattern is generally thought of as a reusable solution to a commonly occurring design problem in object-oriented software. In the seminal work on design patterns [9], written by a group of authors known as the Gang of Four (GOF), patterns are cataloged as:

> "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." [9, p. 3]

They provide solutions at a certain level of granularity, often the class level in a standard OO language. The problems are often centered around how to encapsulate *variability* in a design. Most of the catalogued patterns target as a whole the properties that flexible software are characterized by. Patterns when implemented, often comes with the cost of an extra layer of indirection, pawing the way for increased abstraction in the program.

> "All problems in computer science can be solved by another level of indirection" (Butler Lampson)
>
> "..Except for the problem of too many layers of indirection.." (Kevlin Henney)

Applying several design patterns might create several layers of indirection. This can have a negative impact on performance, but this is seldom the focus of design patterns. Often the focus is *modifiability*; the ease with which the software can accommodate changes.

Design patterns can be derived from a few first principles [9]:

- Program to an interface, not an implementation.
- Favor object composition over class inheritance.

- Consider what should be variable in your design.

The granularity of the first two principles is at the class level. The first principle will help in abstracting away any implementation details of a class, such that clients cannot come to depend on them. By only exposing an interface, the intrinsics of an implementation class is simply hidden, which means that we are free to alter it without fear of affecting clients.

The next principle centers on reuse of classes. There are basically two mechanisms for supporting reuse of classes in mainstream OO: *Inheritance* and *composition*, each with different characteristics.

Inheritance or *white-box reuse* [9, p. 19] allows one to specialize existing behaviour by overriding parts of or extending an existing class. This is expressed quite easily in the language. There are some specific problems with inheritance [9, p. 19]. Inheritance is said to break encapsulation, in a sub/super-class relationship, the subclass can come to inadvertently depend on implementation details of the superclass. A change in the superclass might cause a ripple effect to all dependent subclasses. The severity of this problem depends on the language: Do subclasses have access to private state of superclasses, which access modifiers are present to control this, etc.

If object composition was used instead, there would be no such tight coupling, since all implementation details would be hidden. Composition is defined dynamically at runtime, whereas inheritance is a compiletime construct. This gives us more flexibility in defining the relationships between classes, we can at any point in time change a reference from one object to another, as long as our first principle is in effect. Composition as a reuse mechanism, is coined *black-box reuse*, since the objects in the composition does not have access to each others private state. But, as discussed later, object composition carries it own set of problems.

The last principle captures the *intent* of most design patterns, in that they are a means to capture variability. The two first principles are instrumental in achieving the goal. By using patterns in our design we will be able to change aspects of our software without *redesign*.

2.2 Design space

The design patterns space presented in Table 2.1 has two dimensions: *Purpose* and *scope* [9][p. 10].

Along the dimension of purpose patterns can be classified as being either *creational*, *structural* or *behavioral*. Creational patterns deals with the creation of objects, while structural patterns deals with the composition of classes or objects. Behavioral patterns describe object interaction and often distribution of responsibility between objects.

The dimension of scope specifies “whether the pattern applies primarily to classes or to objects “[9, p.10].

Class patterns deal with relationships between classes and their subclasses. Since these relationship are defined at compile time through inheritance, they are fixed and cannot change at runtime. In class patterns the pattern as a whole is contained in one class hierarchy.

Object patterns on the other hand, deal with relationships between objects, such as composition and delegation. This means that the relationship can change at runtime.

In object patterns the combined behaviour of the pattern is distributed among several objects at runtime.

Purpose / Scope	Creational	Structural	Behavioral
Class	Factory Method	Adapter(class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter(object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Table 2.1: Design pattern space

2.3 Language Features and Patterns

Design Patterns are not language agnostic:

> "One person's pattern can be another person's primitive building block." [9, p. 3]
>
> "The choice of programming language is important because it influences one's point of view ... that choice determines what can and cannot be implemented easily." [9, p. 4]

In some special cases a language even have direct support for the concept embodied in a pattern. Let us take a look at the Singleton pattern [9][p. 127]. The intent of the pattern is:

> "Ensure a class only has one instance, and provide a global point of access to it." [9, p. 127]

As the code below shows, a typical Java implementation could involve a private static field, a private constructor and a fcatory method. Instead of constructing the object with a new expression, the method getInstance() is invoked.

```
public class ClassicSingleton {
  private static ClassicSingleton instance = null;
  private ClassicSingleton() {
    // Exists only to defeat instantiation.
  }
  public static ClassicSingleton getInstance() {
    if(instance == null) {
      instance = new ClassicSingleton();
    }
```

```
        return instance;
    }
}
```

Let us take a look at how the same pattern could be implemented in the programming language Scala [18].

```
object ClassicSingleton {
}
```

An **object** defines a singleton object in Scala. There is only instance of this object in any given program and it can be passed around. It has the same semantics as the Java example and it captures the intent of the pattern. Since we have no logic in the Java class ClassicSingleton there is no logic in the Scala singleton object. There are a lot of variations of the Singleton pattern and some are more easily expressed by extending the boilerplate code available in the Java example, than trying to extend the Scala example. Nevertheless, the example shows how a language construct provides direct support for the central concept of a pattern.

2.4 Qualities Of Design Patterns

Design pattern qualities include

- Capturing knowledge and promoting design reuse.
- Serving as a vehicle of communication and
- Documenting software architecture.

Patterns serve as a vehicle of communication in the sense that they enable a common vocabulary for different stakeholders involved in a given software project. They enable architects and developers to discuss and understand the structures of software, whilst ignoring the itty-gritty details of a specific language such as C++ or Java. Patterns are not language agnostic, but to the mainstream languages often employed in larger projects, they are. In other words they enable communication at a certain level of abstraction which is beneficial for discussing software architecture.

Design patterns also come in handy when documenting software architecture. Given an overview of which and how different patterns are applied to the architecture, enables newcomers to the project to understand the overall architecture. As such, it flattens the otherwise steep learning curve often involved in understanding other peoples source code, especially when dealing with large projects.

It could be argued that the most important quality of design patterns is the encapsulated knowledge it represents. Enabling novices to learn from more experienced developers and apply proven designs.

2.5 Problems with Design Patterns

Several problems concerning the *implementation* of design patterns has been identified [5]:

- Traceability
- Reusability
- Implementation overhead
- Self problem [12]

Traceability concerns the visibility of the pattern in the source code. If a concept of the pattern is not supported directly in the language in which it is expressed, the concept must necessarily be expressed with the available constructs, perhaps mainly *implicit*, through e.g. message sends between collaborating objects. Since the concept is now scattered over the source code, maybe over several source files, it is no longer a conceptual identity, which it was at the design level [5][p. 3]. This problem gets worse when several patterns are applied and their deconstructed concepts are intertwined in the source code.

The *reusabiliy* issue is due to the lack of componentized patterns. The pattern implementations, as presented by GOF, are not reusable from one instantiation of the pattern to the next [5][p. 3]. The pattern is deeply embedded in the specific problem domain.

Implementation overhead is mainly:

> "due to the fact that the software engineer, when implementing a design pattern, often has to implement several methods with only trivial behaviour, e.g. forwarding a message to another object or method." [9, p. 3]

This relates to the languages ability to easily express the concepts involved in the pattern. Another aspect of the implementation overhead problem is that several patterns contains boilerplate code that needs to be rewritten each time the pattern is implemented.

And finally, the *self problem*. The object composition employed in design pattern implementations is most often combined with a sort of delegation, in the mainstream object-oriented (OO) sense. The delegation is implemented with message forwarding, because of the *lack of "true delegation" constructs* in the language itself. This is the cause of the self problem, also known by the more descriptive term *broken delegation*. The self problem is directly related to object patterns (2.2) that uses composition/delegation. When an object receives a message and forwards it to its delegatee via a normal message send, *self* is bound to the delegate and not the intended receiver of the message (delegatee). This has a number of consequences.

Example Lets us see a concrete example in order to illustrate the problems involved with design pattern implementations. The code shown implements a pattern. Implementation details are shown where relevant.

```
public class VectorStack<T> implements Stack<T> {

  private Vector<T> v;
  VectorStack(Vector<T> v) { this.v = v;       }
  public Enumeration<T> elements() { return v.elements(); }
  public boolean isEmpty() { return v.isEmpty(); }
  public T pop() { return v.remove(v.size() - 1); }
```

```
    public void push(T e) {      v.add(e); return this }
    public int maxSize() { return 256; }
  }

  public interface Stack<T> {

    boolean isEmpty();
    Enumeration<T> elements();
    Stack<T> push(T e);
    T pop();
    int maxSize();
  }

  class Vector<T> {

    boolean isEmpty();
    Enumeration<T> elements();
    T remove(int atIndex);
    int size();

    Vector<T> add(T e) {
      if (size() < this.maxSize())
      ...
      return this;
    }

    int maxSize();
  }
```

We can start by asking: What are the roles of VectorStack<T>, Stack<T> and Vector<T>? The question relates to *traceability*. Identifying which pattern and which classes is part of it, and what the roles of the individual classes are, is non trivial. The pattern in this case is Adapter [9][p. 139]. The Stack<T> interface is our target interface. This is the interface that the class *Vector<T>* should conform too. The class VectorStack<T> is the adapter implementation class.

The adapter implementation class is clearly not reusable in the sense that, as soon as, e.g. our target interface changes, which corresponds to another instantiation of the pattern, the entire class would have to be rewritten. This relates to the problem of *reusability*.

An example of *implementation overhead* is the method **void** push(T e) from VectorStack<T>. The method body is just a trivial method invocation.

Three different consequences of the self problem are present. The pattern implementation uses composition and delegation to achieve its goal. The class VectorStack<T> is the delegating class, while Vector<T> is the delegatee.

First we have the problem of *self sends*. The method **int** maxSize() in Vector<T> is used in the Vector<T> add(T e) method. The method maxSize() implements a policy that determines how large the vector is allowed to grow. In our VectorStack<T> we would like to change this policy, but sadly the method **int** maxSize() located in this class does not get called in the implementation of Vector<T> add(T e), since **this** is bound to the delegate, not the object that the message is forwarded to.

Secondly, we have the problem of *identity*. Returning to the Vector<T> add(T e) method, we see that it returns **this**. This is done inorder to promote method chaining. Sadly we must discard the result in **public** Stack<T> push(T e), since **this** is bound to the delegatee and

not the object that originally received the message.

Thirdly, the *trivial forward* problem. The methods **boolean** isEmpty and Enumeration<T> elements, both from VectorStack<T>, would not be needed if "true delegation" was present.

Conceptually, the composite object consisting of a VectorStack<T> and Vector<T> is one, but the implementation language hinders the realization of this. In the Self [26] programming language, "true delegation" is present. One can declare a parent pointer in an object (delegate), that points to another object (delegatee). When the delegate receives a message it will look for a corresponding local method, and invoke the method if a match was found. If no match was found, the method lookup algorithm will traverse all parent pointers and look for a matching method. This takes care of the *trivial forward* problem. If a matching method was found through a parent pointer, the method will be invoked, but self will be bound to the new receiver. This takes care of the *identity* and *self sends* problems. Notice the similarity with multiple inheritance, in that it allows us to deal with the same problems: We can override an, in this case, inherited method. Self is properly bound and no trivial forwards are needed. But compared to classic multiple inheritance, the delegation in Self is dynamic.

When implementing delegation in a language such as Java, we can pass a reference, when invoking methods on the delegatee, that points to the delegate. This gives the delegatee access to the original receiver of the message. But this creates a coupling between delegate and delegatee, resulting in *less reusable classes*.

2.6 Solutions

What might the solutions to the problems mentioned in Section 2.5 be? If patterns were available as components, reusability would not be an issue.

The root cause of lack of traceability seems to be that design patterns are only implicit expressed in source code. The code is scattered across several classes and intertwined with other code. One solution to this problem would be to increase the *locality* of pattern code. Ideally all pattern specific code should be in one place. This is especially hard when patterns have crosscutting concerns.

Having the pattern available as a component would increase traceability, since the usage of the component is apparent in the source code. Note that completely localising pattern code is not the same as expressing the pattern as a reusable component. A completely localized solution written in a language that can express crosscutting concerns, is not necessarily a reusable solution.

Arguably, a language that provides constructs that more easily captures the concepts in patterns, would remove some of the noisy implementation overhead that hinders comprehensibility and traceability. But this is highly subjective, since it depends very much on the programmers knowledge.

Yet another solution, would be to annotate or comment the source code. In the first case, tools can be build that process these annotations and perhaps provide help for the programmer.

Implementation overhead should be significantly lower when using a componentized pattern. A language that has direct support for the concepts involved in a pattern or otherwise provides a more concise implementation of the pattern, will naturally have lower implementation overhead. A builtin pattern, such as singleton in Scala, is an example of this.

The *self problem* is a result of using composition/delegation in a language without true delegation. GOFs second principle promotes composition because of its flexibility and dynamic nature. Avoiding the use of composition/delegation in a pattern if possible, will eliminate the self problem and result in more reusable classes.

2.7 Summary

This chapter has presented design patterns and the principles behind. Several implementation problems were identified, along with possible solutions.

A concrete analysis of the self problem, showed a number of consequences: The problems of trivial forwards, self sends and identity.

This concludes the first of our goals, as stated in Section 1. The following chapter will introduce Scala and its novel constructs.

Chapter 3

Scala

This chapter will introduce Scala and describe the novel features present in the language. We will start with an overview of the language and proceed with traits and mixin composition, this will include a discussion of multiple inheritance and linearization. Further, we describe the unification, especially between functional features and object-orientation that has been performed in Scala. We proceed to explicit self types and their relation with inheritance. Finally, a technique for aspect-oriented programming (AOP) in Scala is presented and we conclude with identifications between component terminology and constructs in the language.

3.1 Overview

Scala is a statically typed multi-paradigm language that attempts to *unify object-oriented and functional programming*. Every value is an object and every operation is a message send [18, p. 1]. Functions are first-class values and pattern matching can be performed on some objects. Methods in classes are latebound and can be overridden and refined in subclasses, overloading is performed based on the list of arguments.

Scala programmes are compiled to bytecodes and runs on the Java Virtual Machine (JVM) while providing full interoperability with Java. This means that a Scala programmer has access to a wide range of third-party libraries besides the Java standard library itself. This is essential, the entry bar for adoption of new languages is higher than ever, since so much depends on the available libraries.

Scala favors an expression oriented programming style that minimizes side-effects, this includes the usage of immutable fields in classes, which are encouraged. For efficiency or programmer comfort an imperative style can be adopted at any time.

An object-oriented version of algebraic data types known as case classes is present. This enables convenient decomposition of class hierarchies [18, p. 13] via pattern matching. Local type inference is present in the language, which means that type annotations are fewer than in conventional OO mainstream languages. Type annotations can always be inserted if needed for documentation purposes.

Everything is nestable, including classes and functions, this idea originates from the BETA [14] language. Nestable classes and functions can improve encapsulation and enables more precise modelling of concepts.

Scala supports functional type abstraction, also known as generics, and abstract types which might be thought of as an object oriented version [18, p. 8]. Upper and lower

bounds can be specified for both kinds of type abstraction. Since the language features subtyping and generics, the interplay can be refined with variance annotations. This enhances reusability in a type-safe way.

Scalas uniform object model is quite similar to Smalltalks[1]. All classes inherit from scala.Any [18, p. 3]. As can be seen in Figure 3.1 the hierarchy has two branches: scala.AnyVal which every *value class* inherits from and scala.Anyref which all *reference classes* inherits from. Since interoperability with Java is needed, every primitive Java type name corresponds to a value class and java.lang.Object maps to scala.AnyRef. Two bottom types exist, scala.Nothing and scala.Null. When interacting with Java Null is needed, and Nothing is convenient when dealing with covariant generic collections such as lists. With the usage of Nothing we can make the empty list a subtype of any other type parameterized list.

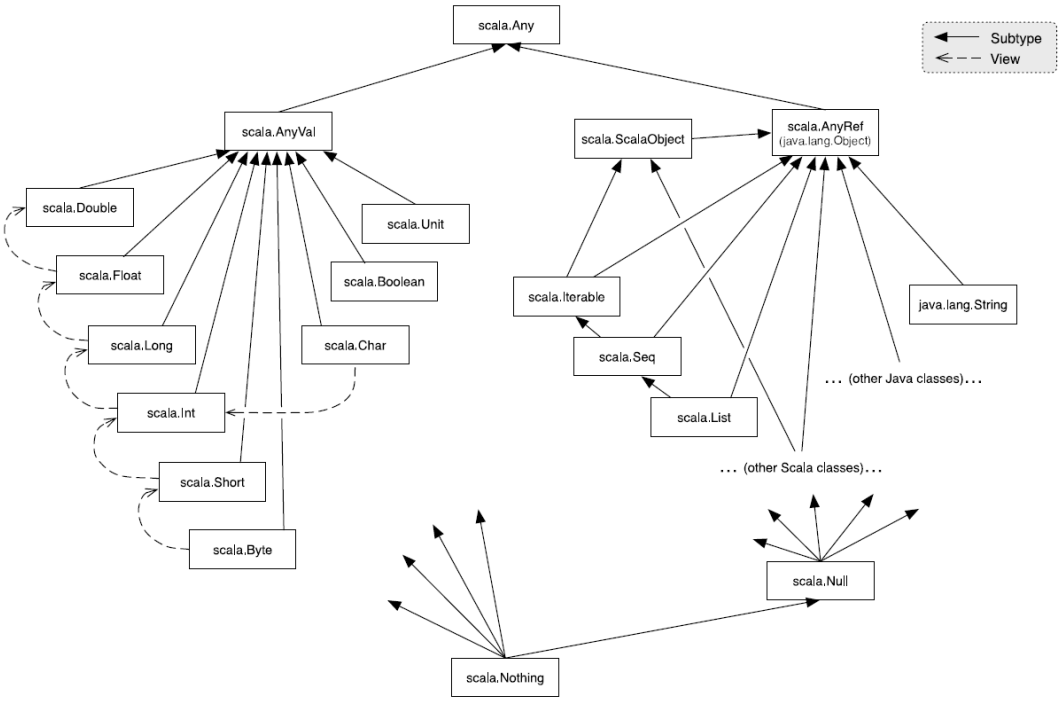

Figure 3.1: Class hierarchy of Scala

The following shows some typical Scala code. The class Rational (adopted from [21][p. 139]) models immutable rational numbers. The class constructor is located just after the class name. The class contains no mutable state, indicated by the use of the **val** (value) keyword for fields, e.g. **val** numer = n / g. Notice that nonstandard characters are allowed when defining methods, such as the method **def** + (that: Rational) : Rational that takes another rational that: Rational as an argument, adds it to self, and returns the result. Type annotations are written *after* the parameter name, or method definition, and are prefixed with :.

```
// Constructor parameters
class Rational(n: Int, d: Int) {
  require(d != 0)
```

[1] http://en.wikipedia.org/wiki/Smalltalk

```
    // Immutable fields
    private val g = gcd(n.abs, d.abs)
    val numer = n / g
    val denom = d / g

    // Method definition
    private def gcd(a: Int, b: Int): Int =
     if (b == 0) a else gcd(b, a % b)

    // Method definition
    def + (that: Rational): Rational =
     new Rational(
     numer * that.denom + that.numer * denom,
     denom * that.denom
    )

    def * (that: Rational): Rational =
     new Rational(numer * that.numer, denom * that.denom)

     override def toString = numer +"/"+ denom
}

// T is a type parameter
class Stack[T] {
  private var elems: List[T] = Nil
  def push(x: T) { elems = x :: elems }
  def top: T = elems.head
  def pop() { elems = elems.tail }
  def exists(p: T => boolean) =
  elems.exists(p)
}
```

The listing also contains a generic class Stack[T]. The stack is implemented with an immutable list **var** elems: List[T] stored in a mutable field, indicated by the use of **var** (variable). Notice the method **def** push(x: T), which concatenates the argument with our current list of elements. Scala does not have operators per se, the :: list operator known from the ML [2] family of functional languages is just a message send. Our stack class also contains a higher-order function **def** exists(p: T=> boolean), indicated by the arrow type signature.

Scala features some novel abstraction and composition mechanisms: Abstract type members, explicit self types and mixin composition. These are all identified as essential when writing components [15]. These language constructs will be further explained and explored in the following, together with the concept of *unification*.

3.2 Traits and Mixin Composition

Traits or *mixins* are essentially abstract classes without constructors. Traits can be used anywhere an abstract class is expected, but only traits can be *mixed in* a class via mixin composition. They can be thought of as *abstract subclasses*. Mixin composition is an extension mechanism comparable to standard class extension mechanisms, such as single

[2] http://en.wikipedia.org/wiki/ML_(programming_language)

inheritance found in Java, although more general [6]. The result of a mixin composition is a new class.

Traits can be used just like interfaces in Java, but can also contain implementation code. Listing 3.1 gives an example. Trait Cell is an interface, describing a cell type, with two abstract members, save and retrieve. The class StandardCell extends it and provides an implementation of the interface. Trait Similar on the other hand contains implementation code for the method isNotSimilar. We can only extend one class, but we can mixin several traits also, as such we have a form of multiple inheritance in Scala [18, p. 11].

Mixing in a trait is done with the keyword **with** at creation time in a **new** expression, or in a class declaration: **with** T_1 ... **with** T_n, where T_i is some trait. The keyword functions as Scalas *mixin composition operator*. Mixing in a trait at creation time is shown at the end of Listing 3.1 in the assignment of **val** cell. The trait Similar is mixed in with StandardCell. An implementation for the isSimilar(x:Int) method is provided at the same time, since all abstract members of the resulting composition must be provided with an implementation before instantiation.

Listing 3.1: Traits

```
trait Cell {
  def save(x:Int)
  def retrieve
}

class StandardCell(protected var state:Int) extends Cell {
  def save(x:Int) = state = x
  def retrieve = state
}

trait Similar {
  def isSimilar(x: Int): Boolean
  def isNotSimilar(x: Int): Boolean = !isSimilar(x)
}

val cell = new StandardCell(3)
  with Similar { def isSimilar(x:Int) = state == x}
```

Mixin composition performed at creation time gives us a convenient way of creating a new anonymous class, that is the result of the mixin operation. It safes us from explicitly textually creating and naming classes, of all the different combinations of traits we could imagine. The mixin composition can be seen as a shorthand for defining and naming a new class, as is done explicitly here instead.

```
class StandardCell(protected var state:Int) extends Cell
 with Similiar {
  def save(x:Int) = state = x
  def retrieve = state
  def isSimilar(x:Int) = state == x
}
```

The type of the resulting mixin composition in Listing 3.1, is a *compound type* StandardCell **with** Similar.

It is, in a sense, a structural type, since we have not named it.

Since it is possible to have implementations of methods and fields in traits, *rich interfaces* [21][p. 249]are a possibility. A rich interface is typically an interface that offers a lot of convenience methods, methods that rely on a few other methods that must be implemented when the trait is mixed in or extended. The method isNotSimilar is a simple example of a convenience method. Sparse interfaces are more common in Java, the implementation burden is enlarged for each method in an interface, since interfaces cannot contain implementation.

3.2.1 Multiple inheritance

Although traditional multiple inheritance is a powerful technique it suffers from the *diamond problem*. Several semantic ambiguities arise when part of the inheritance hierarchy is "diamond shaped". An instance of a problematic hierarchy shape is illustrated in Figure 3.2. Class B and C both inherit from A and D inherits from B and C. If D calls a method originally inherited from A that both B and C overrides, which method should be invoked, the one in B or the one in C ? Which of the definitions should we inherit? Java completely avoids this problem with its single inheritance, its composition mechanism will never allow such diamond shaped hierarchies.

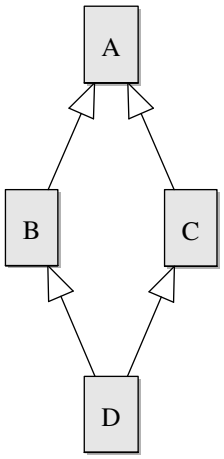

Figure 3.2: Diamond Problem

Other problems caused by multiple inheritance are: How are super calls resolved? What does it mean, in general, to inherit a class by several different paths?

The answers given by the language must for the sake of comprehensibility be fairly straight forward to understand. Otherwise the mechanism will be a source of subtle programming errors. The mixin composition mechanism in Scala alleviates these problems via the process of *linearization*.

3.2.2 Linearization

Scalas answer to the different issues imposed by multiple inheritance is linearization of the class hierarchy. Because of mixins the inheritance relationship on *base classes*, which are those classes that are reachable through transitive closure of the direct inheritance relation [18, p. 11], form a directed acyclic graph, which might include some problematic diamond shapes. The linearization ensures that base classes are inherited only once.

In a mixin composition A **with** B **with** C, class A acts as actual superclass for both mixins B and C. To maintain type soundness, A must be a subclass of the declared superclasses of B and C.

The linearization process can be formalized as follows [15]. Let C be a class with parents C_n **with** ... **with** C_1. The class linearization of C, $\mathscr{L}(C)$ is defined as

$$\mathscr{L}(C) = \{C\} \overrightarrow{+} \mathscr{L}(C_1) \overrightarrow{+} \ldots \overrightarrow{+} \mathscr{L}(C_n) \tag{3.1}$$

The $\overrightarrow{+}$ operator denotes set concatenation where elements of the right operand *replace identical elements* of the left operand.

$$\{a, A\} \overrightarrow{+} B = a, (A \overrightarrow{+} B) \qquad \text{if } a \notin B \tag{3.2}$$

$$= A \overrightarrow{+} B \qquad \text{if } a \in B \tag{3.3}$$

A class can be viewed as a set containing members as elements. A subclass is thus a more specific set, since we may have added more elements to the set. Some of the ambiguities mentioned in Section 3.2.1 occur when two base classes, that is, their defining sets, in the inheritance hierarchy has similar elements. We need to pick exactly one of those elements. The linearization process deals with this by using an overriding strategy when two identical elements are encountered during the linearization.

The overriding strategy can be summarized as follows. A concrete definition always overrides an abstract definition. Secondly, for definitions M and M' which are both concrete or both abstract, M overrides M' if M appears in a class that precedes, in the linearization of C, the class in which M' is defined. Comparing this with our earlier problematic diamond shaped hierarchy (3.2.1), would mean that the method invoked would depend on the order of which D inherits from B and C, D **extends** B **with** C would inherit the method from C, while D **extends** C **with** B would inherit the method from B.

An important aspect of the linearization process is that it is fairly *easy to understand the mechanism when programming*. There are basically two things to remember: The overriding strategy and that in any linearization, a class is always linearized before all of its superclasses and mixed in traits. This means that methods with calls to super are modifying the behaviour of the superclasses and mixed in traits, not the other way around.

Stackable behavior The end of Listing 3.2 shows another example of a mixin composition. The linearization order of the resulting composition is {StandardCell, Max, Doubling, AnyRef, Any} (3.1).

When cell.save(2) is evaluated the call will *follow the linearization order*. The Doubling trait will multiply the argument before calling save on the next trait and so on. The call to **super** in each of the traits are latebound. The consequence is that we have a mechanism to implement *stackable behavior* [21][p. 255]. The order of the mixins in the composition is important, since the call will follow the linearization order, and in this case have different semantics depending on the order. The **abstract** keyword is needed, since we are overriding an abstract method in the trait Cell.

Listing 3.2: Stackable behaviour

```
trait Doubling extends Cell {
  abstract override def save(x:Int) = super.save(2*x)
```

```
}

trait Max extends Cell {
  abstract override def save(x:Int) =
   if( x > 10 ) throw new Exception("overflow!") else super.save(x)
 }

 val cell = new StandardCell(3) with Doubling with Max
 cell.save(2)
```

3.3 Unification

Unification is defined in Merriam-Webster's Dictionary as "the act of unifying, or the state of being unified". Several programming language concepts have been unified in Scala, resulting in increased expressitivity in the language. The following concepts has been unified:

- Functions and classes.
- Modules and objects.
- Algebraic data types and class hierarchies.

The following sections will describe the unifications and how they provide increased expressitivity.

3.3.1 Functions and classes

Every value is an object in Scala and functions are first class values, it follows that *functions are objects*. A function literal such as `(x => x + 1)` is an extension of the trait `Function1[-T1, +R]` shown in Listing 3.3.

The argument type `T1` and return type `R` are both variance annotated, `-` signifies a contravariant position while `+` is a covariant position, which gives a safe version of function subtyping.

Listing 3.3: Unification of functions and classes

```
trait Function1[-T1, +R] extends AnyRef { self =>
  def apply(v1:T1): R
  ...
}
class Inc extends Function1[Int, Int] {
  def apply(x:Int):Int = x + 1
}

val inc = new Inc
inc(3)
```

The object `inc` is used as a function, it is invoked. The Scala compiler will expand `inc(3)` to `inc.apply(3)` and invoke the `apply` method[18, p. 6], in other words `inc(3)` is just syntactic sugar. It is worth noticing here that *methods and method invocation* still exist as a

primitives in the language. The function is wrapped in an object and can thus be passed around and because of the syntactic sugar we can invoke it easily. Any specific method in a class can be passed around to a higher order function, i.e. another function (or method) expecting another function.

```
class Foo {
  var state = 1
  def printState() = println(this.state)
}

def invokeFunction(f: () => Unit) = f()

def main(args:Array[String]) {
  val foo:Foo = new Foo
  val f: (() => Unit) = foo.printState
  foo.state = 2
  invokeFunction(f)
  foo.state = 3
  invokeFunction(f)
}
___
2
3
```

The method printState is lifted from the foo object and saved in f, i.e. **val** f: (() => Unit) = foo.printState. The function is then passed to a higher-order function invokeFunction that invokes it. Notice how we change the state in the foo object, this is visible to the lifted method.

The close connection between functions and classes is not something new, it dates back to the BETA [14] language, with the unification of classes and functions, amongst others, in the *pattern* construct. The intuition stems from the close resemblance between function activation records and classes. Functions activation records have local definitions, so do objects, in form of fields. Objects are "activation records that survive". In [25], Torgersen talks about *substance* and *activity*. Substance is typically identified with classes, and activity with functions. But a function needs substance in form of a activation record, and an object needs *creation code*, which is activity. The two concepts are intertwined, one rarely goes with out the other. To unify classes and functions they need to share the same mechanisms: Function invocation is thus expressed as class instantiation. Parameters and return result is expressed by fields in an object. This is also how the *v*Obj calculus [20], the theoretical base of several of Scalas constructs, handles instantiation and function invocation.

The unification is not complete in Scala, but we still get some of the benefits of the full unification. The language constructs involving classes, are applicable on functions (when represented as classes), paving the way for increased expressitivity. An compelling example is from the Scala standard library mentioned in [18, p. 6]. Arrays in Scala are treated as special functions over the integer domain, that is Array[T] inherits from Function1 [Int, T]. The apply method **def** apply(index: Int):T sets or retrieves a specific index, allowing for an economic syntax, e.g. someArray(2)=3. Furthermore, we have extended the Function trait with methods for array length, traversal etc. Part of the class can be seen here.

```
package scala
class Array[T] extends Function1[int, T]
 with Seq[T] {
   def apply(index: int): T = ...
   def update(index: int, elem: T): unit= ...
   def length: int = ...
   def exists(p: T => boolean): boolean = ...
   def forall(p: T => boolean): boolean = ...
   ...
 }
```

The main result of the unification is that it gives us first-class functions in an object-oriented setting. We can extend them, pass them around as objects, write higher-order functions and lift any existing method in a class and pass that around as a function.

3.3.2 ADTs and Class Hierarchies

Another unification in the language is of algebraic data types and class hierarchies. Instead of adding algebraic types to the core language, Scala enhances the class abstraction mechanism to simplify the construction of structured data [18, p. 14]. The following shows a simple Standard ML [3] (SML) example of specifying structural data with an algebraic datatype. The recursive function sumtree decomposes the tree structure with pattern matching and returns the sum of all nodes in the tree.

```
datatype tree = Leaf of int | Node of (tree * int * tree);

val t = Node( Node (Leaf 2,3,Leaf 10), 3, Leaf 1);

fun sumtree (Leaf n) = n
  | sumtree (Node(t1,n,t2)) = sumtree t1 + n + sumtree t2;
```

An equivalent example in Scala, is shown here.

```
abstract case class Tree
case class Node(t1:Tree,n:Int,t2:Tree) extends Tree
case class Leaf(n:Int) extends Tree

val tree = Node( Node (Leaf (2),3,Leaf (10)), 3, Leaf(1))

def sumTree(t:Tree):Int = t match {
  case Leaf(n) => n
  case Node(t1,n,t2) => sumTree(t1) + n + sumTree(t2)
}
```

Prefixing a class definition with the **case** keyword implicitly adds factory methods for creation of the class, i.e. we do not have to use the **new** keyword to create instances. This can be seen in the definition of **val** tree. More importantly, we can now pattern match on the class constructors via a **case** expression, e.g. **case** Leaf(n) => n, this is done in an enclosing **match** expression. The constructor parameters exists as value fields in the class. The

[3] http://www.smlnj.org/

variable n is bound to the field n of the Leaf object, and can be used on the right-hand side of =>. We have thus decomposed the Leaf object in a type-safe way. Patterns in Scala can include guards, wildcards that match anything, nested patterns, literals and more.

Our case classes behave as normal classes otherwise, we can have methods etc., in contrary to the algebraic datatypes in ML.

There are two basic problems with case classes. First, since it is possible to arbitrarely extend a case class in modules outside our control, we can not be sure that we have made an exhaustive match in a **match** expression. We can provide a default match with the wildcard pattern _, but there might not exist a reasonable way of handling a default case. A class can be marked **sealed**, which asserts that it is only inherited from in the same source file. This enables the compiler to emit a warning if a match is not exhaustive.

The second problem with case classes is that part of the internals of the class is exposed. We loose representation independence of the constructor parameters, which as mentioned, implicitly exists as fields in the class. If this is a problem, or we wish to pattern match on an existing class that we do not have source code access to, we can use *extractors* instead. An alternative to the case class Leaf can be implemented like this.

```
trait Leaf extends Tree {
  def n:Int
}

class LeafImpl(private val d:Double) extends Leaf {
  def n:Int = d.toInt
}

object Leaf {
  def unapply(leaf:Leaf):Option[Int] = Some(leaf.n)
}
```

The rest of the implementation from the previous example, remains intact.

A singleton object (**object** Leaf) can share the same name with a class, and when it does, the singleton is called the class's companion object. The Scala compiler transforms the fields and methods of a singleton object to static fields and methods of the resulting binary Java class.

The trait Leaf is exposed to clients, whereas the implementation class LeafImpl, is not. The unapply(leaf:Leaf) method in **object** Leaf functions as an extractor, or decomposer, that is automatically called when **case** Leaf(n) => n is encountered in the sumTree method. An optional apply method can be provided in Leaf, this should function as a factory method.

Sadly we loose the compilers ability to check for exhaustive matches when using extractors instead of case classes.

3.3.3 Modules and Objects

Another important unification in Scala, is the equality drawn between *modules and objects*. Modules are basically a way to organize larger programs. The exact nature of a module is language dependent, different languages have different module systems. A characterization would be that a module is a smaller program piece that has a well defined interface and a hidden implementation. A complete program consists of several such modules.

Essential to Java is the concept of *packages*. Packages provides a unique namespace for the types it contains and classes in the same package can access each others protected members. Packages are often used as a way to organize classes belonging to some conceptual entity, thereby structuring ones program. Packages have several shortcomings, you cannot parameterize a package, there is no way to abstract over packages and you cannot inherit between packages. These are all well known properties of classes.

In Java, packages provide a unique namespace for its types, and since we are dealing with a nominal type system, two identical classes in two different packages are two different types. This is also the case with classes contained in objects in Scala, remember that everything is nestable.

Listing 3.4: Objects are modules

```
object moduleA {
  class A
}

class ModuleB {
  class A
}

def main(args:Array[String]) = {
  val moduleB = new ModuleB
  val a1 = new moduleA.A
  val a2 = new moduleB.A
  println(a1.getClass == a1.getClass)
  println(a1.getClass == a2.getClass)
  ()
}
---
true
false
```

In Listing 3.4 two identical classes named A, one located in an object the other in a class are instantiated. The class located in moduleA can be instantiated directly with the correct path, that is **new** moduleA.A. In order to instantiate the one located in **class** ModuleB we need to first create an instance of the class, this allows us to parameterize the resulting module, and then create our object with the correct path **new** moduleB.A.

Both are examples of *path-dependent types*. The method getClass returns the runtime class when invoked, as can be seen a1 and a2 has distinct types. More precisely moduleA .A and moduleB.A. The exact type depends on the *runtime identity of the actual objects involved in the path*. The path must be immutable in order to secure type-safety. This is guaranteed with use of the **val** keyword.

Path-dependent types are essential in the unification of objects and modules. Programmers would expect that two identical classes, but located in two different namespaces are distinct types in a nominal type system. Path-dependence insures this.

Since classes also serve as object generators and we have equated objects with modules, that are instantiations of classes, our modules are more expressive than standard Java packages. We can inherit between modules, we can parameterize them and so forth.

3.4 Abstract Types

Abstract types are known from SML modules. Modules in the SML sense are a powerful construct for encapsulation, abstraction and reuse of code. *Structures* provide an *interface* or signature to the surrounding world, thereby hiding the implementation of the individual elements in the module body, such as datatypes and functions. Different signature can be combined with a module which gives a fine grained control over the visibility of names and types. Compare this to the coarser modes of abstraction often provided by mainstream OO languages such as Java, with the keywords *private* and *protected*, which only limits the scope of specific fields and methods. In SML structures it is possible to *abstract over types*, lets look at an example.

```
signature QUEUE =
  sig
  type 'a queue
  exception Empty
  val empty : 'a queue
  val insert : 'a * 'a queue -> 'a queue
  val remove : 'a queue -> 'a * 'a queue
end

structure Queue :> QUEUE =
  struct
  type 'a queue = 'a list * 'a list
  val empty = (nil, nil)
  fun insert (x, (bs, fs)) = (x::bs, fs)
  exception Empty
  fun remove (nil, nil) = raise Empty
    | remove (bs, f::fs) = (f, (bs, fs))
    | remove (bs, nil) = remove (nil, rev bs)
end
```

The type of 'a Queue.queue is abstract, operations of the queue, i.e. empty,insert,remove may only be performed on values of this type. The consequence is that all clients of the Queue structure are insulated from the details of how queues are implemented, the concrete type is abstracted away. In the Queue structure a concrete type is specified which is then used by the implementation. In this case the concrete type is a pair of lists. Even though this is the case it is not possible to use a pair of lists from the client side, since we in the signature have specified that the operations are using the abstract type. Another benefit is that we can not *mix*, by accident, elements of two stacks implementing the same signature. This is a problem with Java interfaces, where we might end up mixing implementations of the same interface unintentionally. As the following shows, in Java, cows might accidentally eat fish (the example is adopted from [21][p. 448]).

```
package farm;

public interface Food {}

public interface Animal {
  void eat(Food food);
}
public class Grass implements Food {}
```

```
public class Fish implements Food {}

public class Cow implements Animal {
  @Override
  public void eat(Food food) {}

  public static void main(String[] args) {
    Cow c = new Cow();
    c.eat(new Grass());
    c.eat(new Fish());   // not intended!
  }
}
```

A first attempt to stop Java cows from eating fish, might be to try and specialize the eat method in the Cow class. E.g. **public void** eat(Grass grass), this is not possible since we would not be overriding the method but instead defining a new one. If it was possible in Java it would be unsafe, since it is unsafe to override a method argument covariantly. We need to take a step back and redo the modelling, this time using abstract types in Scala.

```
object Farm {
  trait Food
  trait Animal {
    type F <: Food
    def eat(food:F) = println("Yummi!")
  }

  class Grass extends Food
  class Fish extends Food

  class Cow extends Animal {
    type F = Grass
  }

  def main(args:Array[String]) = {
    val c = new Cow
    c.eat(new Fish) // illegal!
  }
}
```

We use an abstract type member with an upper bound Food, **type** F <: Food in the trait Animal. The upper bound ensures that F will be some subtype of Food. The method eat(food :F) refers only to the abstract type, which will be bound to a specific type later.

In the Cow class, we specify the exact type, we "tie the knot", this is necessary since we cannot instantiate a class with an abstract member. With the abstract type we have specified that the kind of food suitable depends on the animal, in this case grass, which is exactly what we were trying to model.

3.4.1 Family Polymorphism

Abstract type members and path-dependent types enable *Family polymorphism* [8]. In order to understand the concept, we can ask the question: How can a *group of interfaces* be implemented in such a way that their abstractions are not mixed by accident with other implementations? The problem is described in a OO modelling setting in [8]. The

reader is asked to imagine a hotel lobby where several families are located, presumable all waiting for rooms. The receptionist is interested in assigning each *family* to a certain room, while not getting the families mixed up.. Each role in the family is equated with an interface. Family polymorphism enables us to express and operate on groups, or families, of interrelated objects, in a type safe way. Abstract types plays an essential role. Listing 3.5 gives an example in Scala.

Listing 3.5: Family polymorphism in Scala

```
trait Family {
  type M <: Mother
  type F <: Father
  type C <: Child

  class Father(val name:String) {
    def kiss(m:M) =
    println("Showing␣signs␣of␣affection␣towards␣" + m.name)
  }
  class Mother(val name:String)
  class Child(val name:String) {
    def askForhelp(m:M) = println("Screeaaaaming␣at␣" + m.name)
  }
}

object UpperClassFamily extends Family {
  type F = Father; type M = Mother; type C = PoliteChild

  class Mother(name:String, val lastName:String) extends super.Mother(name)
  class PoliteChild(name:String) extends Child(name) {
    override def askForhelp(m:M) =
    println("Asking␣" + m.name + m.lastName + "␣for␣help")
  }
}

object StandardFamily extends Family {
  type F = Father ; type M = Mother; type C = Child
}

def assignFamily(f:Family) = ()

val father = new StandardFamily.Father("John")
val upperClassMother = new UpperClassFamily.Mother("Dorthea␣III")
father.kiss(upperClassMother) // illegal!
```

We model a family as a strict nucleus consisting of a father, mother and a single child, based on our conservative world view and fear of overpopulation. The method assignFamily(f:Family) serves to illustrate the polymorphic aspect, i.e. it's possible to operate on a whole family at a time. We could get close to this just by using standard OO techniques, or perhaps by using the inner classes functionality of Java, but we have no way of expressing the *dependence* of one family member with another. In the Scala example father will only kiss the mother belonging to this *instance* of a family. This is not directly expressible in Java.

Family polymorphism as a programming style supports reusable mutually recursive classes, that can vary together covariantly. Take PoliteChild as an example, it is an extension

of Child, overriding the method askForHelp. The new method uses an extended version of the Mother class. In Java we would have to create a new Father class also, since the father class would refer to the old mother and child classes in askYourMother.

3.5 Self Types

In a class or trait declaration C, an *explicit self type* can be given: Class A {**this**: T => }, where **this**: T => is the self type declaration and T is the self type. It is possible for T to be a compound type, e.g. T1 **with** T2 **with** T3. If a self type is given, it is taken as the type of **this** inside the class, otherwise the self type is taken to be the type of the class itself. Any identifier instead of **this** can be used as an alias, which might be convenient when dealing with nested classed. The self type specifies that the class or trait when created is guaranteed to be mixed in with T, it *requires* T, i.e. when the class is created it will be a subtype of T. In the class declaration the entire namespace, that is, methods, fields etc. of T are available, just as normal inheritance would provide. Compared to inheritance where we extend a *specific* class, an explicit self type states that we will either be extended by T (or any subtype) or be part of a mixin composition with T. This lifts the dependence on an exact class.

In order to ensure type-safety the self type of a class must be a subtype of the self types of all its base classes (3.2.2). In the **new** expression it is checked that the self type of the class is a supertype of the type of the actual object being created. [18, p. 10]

Self types are often used when family polymorphism is combined with explicit references to self [15][p. 7]. Consider the abstract class Graph (example is from [15]).

```
abstract class Graph {
  type Node <: BaseNode;
  class BaseNode {
    this: Node =>
    def connectWith(n: Node): Edge =
    new Edge(this, n); // illegal without self type annotation!
  }
  class Edge(from: Node, to: Node) {
    def source() = from;
    def target() = to;
  }
}
```

The class models a graph data structure, with nodes and edges. We have an abstract type Node with upper bound BaseNode. Class Edge refers to Node, the intention is that when we extend our data structure, and perhaps refine class BaseNode, class Edge will still work with the new version, which enhances reuse. But there is a problem with the method connectWith. The method is there because we want subclasses of BaseNode to support it. Without the self type annotation the type of **this** in class Basenode is BaseNode and not the required type Node. We can solve the problem with an explicit self type **this**: Node =>, which is already done. The class BaseNode will always be a subtype of Node, this is guaranteed by the upper bound.

Explicit self types where mainly introduced for technical reasons in the νObj calculus [20] and were only included in Scala because they seemed essential when family polymorphism is combined with self references, as the above example illustrates. It was later

discovered that their real power was in expressing circular or recursive dependencies between modules or classes and lifting static systems to component based systems. [15, p. 7].

3.5.1 Self type vs Extends

Self types and inheritance are closely connected. So it is worth discussing their interaction and pros and cons

When defining a class hierarchy and using a self type anywhere in the hierarchy, it will result in the need for re-affirming the annotation again for every subclass. This is because self types are not inherited. The code below shows that trait C must contain a seltype annotation in order to compile.

```
trait A
trait B { this: A => }
                    // necessary!
trait C extends B { this: A => }
```

This is more verbose than equivalent code using only normal inheritance with the **extends** keyword. Another limitation is that the visibility of the self type is limited to the enclosing class or trait. The following example is continued from the last. No coercion is performed on b in the call to foo.

```
trait D {
  val b:B
  val test = foo(b) // Error!
}
def foo(a:A) = ()
```

As mentioned earlier self types can be used to express circular dependencies, which inheritance cannot.

```
trait X { this:Y => }
trait Y { this:X => }

// illegal cyclic reference
trait X extends Y
trait Y extends X
```

This is an important aspect of self types since modules are unified with objects. Another thing to consider when deciding whether to use self types or standard inheritance is the resulting linearization order. Inheritance is less flexible here than self types, since the order is predefined in the class declaration. Whereas using self types the order can be decided in the **new** expression. The last thing to consider is that we cannot implement stackable behaviour with self types, **super** is not bound to the self type, since we are not inheriting from it.

3.6 Aspect-oriented Programming in Scala

Aspect-oriented programming is a technique for separating the different cross-cutting concerns in a system, such as logging, synchronization etc., that would otherwise be intertwined with business logic. The separation enhances modularity. The concerns are cross-cutting because they span multiple abstractions in a program, such as classes or modules. This has a negative impact on maintainability, and makes the system much harder to understand and reason about.

Logging is the prototypical example of a crosscutting concern. An implementation without separation would mean that the logging code is scattered amongst all the classes which needs the functionality.

AspectJ [4] which is an AOP implementation for Java, has the notion of *Advice*. Advice is specific, additional behavior that is executed at certain *join points* in the program. A *pointcut* query detects whether a given join point matches, if it does the specified advice will be executed at the point in the program the join point specifies. These concepts can be used in what AspectJ refers to as an aspect. An aspect is a module, where cross-cutting concerns are collected. Thereby modularizing a part of the system that would otherwise be intertwined with the business logic.

In Scala there are some possibilities for adopting an aspect-oriented programming style.

Listing 3.6 illustrates a logging aspect that uses before and after advice. Our join points are limited to specific methods in the extended class or trait. All logging code is *located in one module*. By using mixin composition we can simulate non-intrusive advice.

Listing 3.6: Advice in Scala

```
trait Channel {
  def send(x:String) = println(x)
}

object LogAspect {

  trait LogAfter extends Channel {
    // before advice
    abstract override def send(x:String) = { log() ; super.send(x) }
  }
  trait LogBefore extends Channel {
    // after advice
    abstract override def send(x:String) = { super.send(x) ; log() }
  }
  def log() = println("logging!")
}

def main(args: Array[String]) = {
      val channel  = new Channel with LogAspect.LogBefore
      channel.send("message")
}
---
logging!
message
```

[4] http://eclipse.org/aspectj/

3.7 Components in Scala

A component is a program part that is to be combined with other parts in an application. Components should by definition be reusable. A component typically has an interface describing the services it provide, another aspect is the services that is required by the component itself.

A problem with current mainstream OO languages is that they have no way of *specifying or abstracting over required services* [16][p. 1]. This makes a specific component harder to reuse in a new context without modifying the component itself.

In Scala we can identify the following equalities between component terminology and language features [17][p. 27].

- Component = class.
- Runtime component = object .
- Interface = abstract class (or trait).
- Required component = abstract member or explicit self type.
- Composition = mixin composition.

The identification of 'Runtime component' with 'object' is a result of the unification of objects and modules. The class construct is more expressive in Scala, thanks in part to nesting. Top-level classes can contain other classes etc, akin to Java packages. What is really new here, is the 'required component' part, which we can specify with either abstract members or explicit self types. In a component composition all requirements of individual components are satisfied, since a concrete member must be supplied for all abstract members in a mixin composition. Using explicit self types we can write components that are recursively dependent on each other (3.5), something that we cannot express with normal inheritance.

Abstract type members and Self type annotations both *minimize hard references between components*. If hard references are present, the component cannot be reused in a context that refines these references.

In the design pattern analysis we will have a special focus on the above constructs, since they carry promises of new ways of writing components, which design pattern are not.

3.8 Summary

This chapter concludes our second goal(1). The novel constructs central for component writing: Mixin composition, abstract types and self types, was presented. The concept of familiy polymorphism, which relates to abstract types, was introduced as a valuable programming technique. An analysis of self types resulted in the observations: Self types are not inherited, they must be repeated in subclasses. They can express circular dependencies, which inheritance cannot. The visibility of a self type is limited to the enclosing

class. And regarding linearization, self types gives us more flexibility in expressing the resulting class hierarchy compared to inheritance.

Three unifications was identified: Functions and classes, ADTs and class hierarchies and modules and objects.

Regarding AOP in Scala, we can implement advice with the technique of stackable behaviour (3.2.2). The next chapter will start by describing the analysis that is to be performed on our catalog of patterns.

Part II

Analysis

Chapter 4

Design Pattern Analysis Overview

This chapter will lay the foundation for the analysis that will be performed of the GOF patterns in the *context of Scala*. Basically we are interested in dealing with the current problems (2.5) design pattern implementations exhibit and exploring whether Scala offers increased expressivity in implementing patterns.

4.1 The Individual Pattern Analysis

Each individual pattern analysis is conceptually divided into two parts, a GOF part and a Scala part. The GOF part introduces the pattern and discusses points of interest. In the context of Scala we analyse the pattern in order to provide an improved implementation or a reusable component. In the process the following will be answered:

- Can and should we componentize this pattern? If not, why?
- Which problems with the original solution can we deal with and what are the enabling features?

Based on the above answers we present a new solution in Scala. The solution must be type-safe, which means that no casts or use of reflection is allowed.

This should conclude our third goal (1).

4.2 Process of Componentization

This section will discuss the things that must be considered when trying to componentize a pattern. In order to fully componentize a pattern we must first be able to identify what varies from one instantiation of the pattern to the next. Our language must be able to abstract over this, or else there is no hope for writing a reusable pattern. Secondly, we must be able to modularize the things that does not vary from one instantiation to the next. This is just as essential.

Summarizing, we must be able to:

- Find what varies in the pattern, and
- abstract over it.
- Find the things that does not vary, and

- modularize it.

This will depend on the exact implementation we have in mind. This is the basic recipe for componentization, but should we always aim for componentization? What qualifies as a successful pattern componentization? First of all the intent of the pattern must be captured, or at least a substantial part of it. This is the part that is hard to formalize, and must be judged individually in each case. Patterns exist in different variations and it might be reasonable to capture just one version.

A second criteria which we call the *less tokens expended* rule is of an objective nature. In using the component less tokens in the source code must be expended than a standard implementation of the same pattern would require, or else it not a successful/useful abstraction of the pattern. Using an abstraction should not result in writing more code, than writing equivalent functionality oneself. This means that there must be a certain amount of boilerplate code in the pattern to modularize, in order for their being a chance of successful componentization, i.e. providing a useful abstraction.

An example, although a bit contrived, can be seen below.

```
// The abstraction
object module {
  def add(a:Int, b:Int) = a + b
}

// Usage of abstraction
import module._
add(1,1)

// which is equivalent to
1 + 1
```

The abstraction `add` is a useless abstraction according to the rule. In using the abstraction we need to write more tokens, than the equivalent `1 + 1`.

4.3 Summary

An overview of the analysis that is to be performed on all individual patterns was presented. The analysis must deal with the issue of componentization and where possible produce an improved solution. A high level view of the process of componentization was presented, along with the less tokens expended rule. The rule is instrumental in judging the usefulness of a component or abstraction.

Chapter 5

Creational Patterns

This chapter will analyse all individual creational patterns (2.2) and if possible provide a componentized pattern or an improved implementation.

5.1 Abstract Factory

Description The intent of the pattern is:

> "Provide an interface for creating families of related or dependent objects without specifying the concrete classes." [9][p. 87]

We start with a motivating example [9][p. 87-88]. Say you are interested in building a GUI library framework that should run on different platforms, such as Microsoft Windows or Mac OSX. In order to support a native look and feel on each platform, e.g. for a graphical window, while maintaining portability of client code across the platforms, only the interface of the window should be exposed to the clients. Typically a window is composed of several widgets, some of these might be platform specific. Exactly which and how a concrete window is created, is abstracted away from the client code by an *abstract factory*. This enables us to change the exact *creation code* with out changing the clients. Since the client is unaware of the concrete classes, no platform dependencies are introduced in the client. We can seemingly change the implementations of the window and the associated widgets and provide new ones, as long as they adhere to the abstract interface. Typically a factory creates a *family* of *products*, in our GUI setting, this might be windows, menubars etc. An important issue is that we should not mix the platform dependent classes [9][p. 88], as this might lead to runtime errors.

Concrete factories are most often implemented as singletons [9][p. 90].

Analysis

In the original solution there are no static guarantees that products from different factories will not get mixed. This might be a problem when products depend on each other, e.g. A CreateProductA(B). Only the programming discipline surrounding the use of the factory ensures that products will not get mixed. Expressing product inter-dependence can be done with abstract type members, which in theory will create more opportunities for reuse in the descendants of a specific factory.

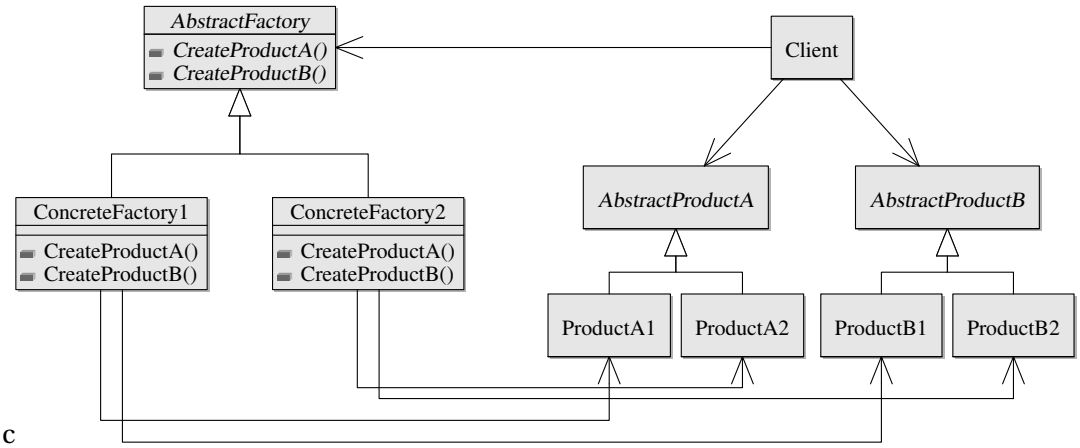

Figure 5.1: Abstract factory pattern UML

If we view the products from a specific factory as a *family of products*, it is clear that the pattern might benefit from the techniques used in Scalas version of family polymorphism (3.4.1).

Encapsulation of concrete types can be performed with nesting of classes and access modifiers. Creation code will be encapsulated inside the factory.

Componentization The things that vary from one instantiation to another are

- Type of products.
- Number of products.
- Creation code.

The type of a product could be abstracted away with an abstract type member. Creation code could be represented by a first-class function that is passed to the factory. Sadly we cannot abstract over the number of products. This could be solved by providing several abstract factory classes, where they only differ in the number of products.

With the above solution we loose the ability to express product interdependence and we loose the ability to encapsulate the concrete types in a clean way with nesting. And we gain very little, as can be seen in the following example, which shows a reusable factory for two products. Besides failing to abstract over the number of products, the trait will most likely not survive the less tokens expended rule.

```
  abstract class Factory {
    type T1
    type T2

    def createA(f:() => T1)
    def createB(f:() => T2)
  }
}
```

Scala Solution

The trait WindowFactory in Listing 5.1 defines the interface for a factory. It contains the type members aWindow and aScrollbar. These type members are upper bounded by our abstract products and must be refined in concrete factories. Instantiation code common for concrete factories can be reused in subclasses as long as they only refer to the abstract types defined. The abstract products are both abstract classes nested in the factory trait, that is, class Window and class Scrollbar. Any concrete product must extend either. The abstract factory methods createWindow and createScrollbar hides the actual instantiation code from clients.

Listing 5.1: Abstract Factory trait

```
trait WindowFactory {
  type aWindow <: Window
  type aScrollbar <: Scrollbar

  def createWindow(s:aScrollbar)
  def createScrollbar()

  abstract class Window(s:aScrollbar)
  abstract class Scrollbar
}
```

The following listing shows how we can extend our abstract factory with a concrete factory. Our concrete factory is a singleton object containing protected nested classes. These are the concrete products. Since they are protected they are hidden from clients.

Listing 5.2: Abstract Factory example usage

```
object VistaFactory extends WindowFactory {
  type aWindow = VistaWindow
  type aScrollbar = VistaScrollbar

  def createWindow(s:aScrollbar) = new VistaWindow(s)
  def createScrollbar() = new VistaScrollbar

  val window:aWindow = new VistaWindow(scrollbar)
  val scrollbar:aScrollbar = new VistaScrollbar

  protected class VistaWindow(s:aScrollbar) extends Window(s)
  protected class VistaScrollbar extends Scrollbar
}
```

For maintenance reasons we might not be interested in having the actual source code implementing the product classes located inside the factory. Using an explicit self type, we can express the dependence that exists between a module that provides product classes and any WindowFactory. This enables us to extend Window and Scrollbar in a module where they otherwise would not be in scope.

```
trait VistaWidgets {
  self:WindowFactory =>
  protected class VistaWindow(s:aScrollbar) extends Window(s)
```

```
    protected class VistaScrollbar extends Scrollbar
}
```

Summary Nesting of classes is useful for encapsulation of specific types. The factory is essentially just a namespace trait, or repository, for the concrete products. Abstract types enables us to express product-interdependence. This gives us static guarantees that products from different factories cannot be mixed. Furthermore, they enhance the opportunities for reuse in factory subclasses.

Summarising on the qualities not present in the original GOF solution.

- Product-interdependence: Impossible to mix products from different factories.
- Singleton factories are trivial to implement.
- Nested product classes: Implementation classes are easily hidden from clients.

5.2 Builder

Description The intent of the Builder pattern:

> "Separate the construction of a complex object from its representation so that the same construction process can create different representations." [9, p. 97]

The motivating example from GOF involves a system containing a reader for the RTF document exchange format that can convert the input to several different representations, such as ASCII text or GUI widgets letting the user edit and see the text [9, p. 97]. Figure 5.2 shows UML for the pattern. Relating the example to the figure, our reader plays the role of the Director. Different builders exist for each product, the products are in our case the different representations. Using a concrete builder the director builds each part of the product and collects the final result with GetResult. During the build process the director provides the builder with the needed input for construction of the product. In our case the reader reads characters, in tokens, from an input stream and calls the appropriate BuildPart on the builder, depending on the token.

As is intent of the pattern, the builder encapsulates the construction, hiding internal data structures of a specific representation, while allowing the director to use the same construction process to create different representations. Typically no abstract product exist since the representations can differ wildly.

Analysis

Builders intent is to encapsulate the process of building itself, a fitting abstraction mechanism for achieving this is *methods*. The outcome of the build process differs in great detail, only the input stays in general the same between different concrete builders. As such, Scala offers nothing new in capturing these concepts, methods are a good fit. We could try to express interdependencies of the build process with abstract types, but since the output differs wildly it does not make much sense.

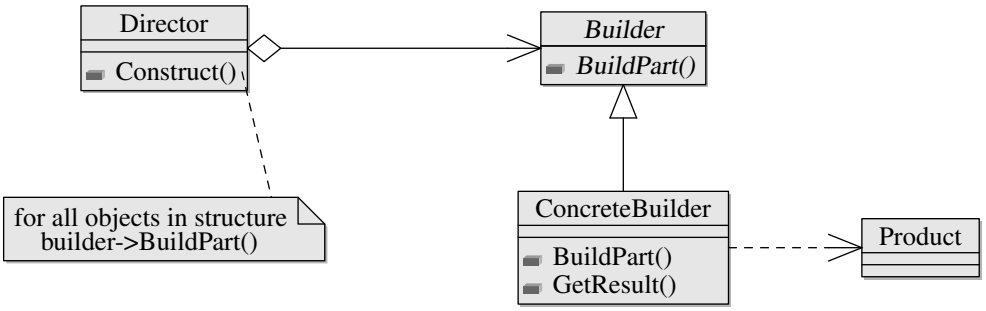

Figure 5.2: Builder pattern UML

The original code for the motivating example would benefit from *double-dispatch*. The director class contains a **switch** statement that branches on the type of input, depending on the type a specific build method is invoked in the branch. The controlling expression in the **switch** statement, i.e. `t.Type` as seen in Listing 5.2, must be of an integral type, so it not the actual type that is used, but a property of the object `t`.

With double-dispatch we could avoid this switch statement, the actual runtime type of the input would decide which method was called, thereby simplifying the code.

```
// original
switch t.Type {
  CHAR:
    builder->ConvertCharacter(t.Char);
  FONT:
    builder->ConvertFontChange(t.Font);
 }

 // hypothetical multiple dispatch
 builder->Convert(t)
```

This would off course require that we had 2 identically named `Convert` methods each taking one parameter with distinct types. Note that the last `t` is different from the first `t`.

We can simulate multiple dispatch with pattern matching.

```
t match {
  case c @ Character() => convert(c)
  case f @ Font() => convert(f)
}

def convert(t:Font) = println("Converting font")
def convert(t:Character) = println("Converting character")
```

The solution using pattern matching matches on the actual runtime type, and not some member of the class used for describing the type. Though not as concise as our hypothetical `builder->Convert(t)`, it is superior than the original.

Componentization The things that differ between instantiations are the set of encapsulated build methods in an builder class. So does the resulting product and the input to each build method. It is not possible to abstract over a set of methods in a class in a

meaningful way, this implies that we cannot modularize a builder. Since all other roles depend on the details of the builder we cannot provide any component.

Summary The builder solution as presented by GOF would benefit from double-dispatch, but this is not directly present in Scala. The analyses showed that double-dispatch has a connection too, and can be simulated with, pattern matching.

Compared to mainstream OO languages Scala does not provide any new constructs that are otherwise beneficial for the pattern.

5.3 Factory Method

Description Intent of the Factory Method pattern:

> "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses." [9, p. 107]

Factory methods are typically used when a class cannot anticipate the class of objects it must create. A motivating example is in the context of an application framework [9, p.107-8]. The abstract application class is capable of creating documents. An abstract document class is subclassed by users of the framework depending on the exact application nature. The framework creators cannot anticipate exactly which document subclass needs to be created by the application class, since the document class can be arbitrarily extended. Instead a method in the application class is used as a factory method: It returns some instance of a subclass of the abstract document class. Exactly which, can be refined in subclasses of the application class by overriding the factory method.

The document class plays the role of Product as seen in Figure 5.3. Concrete documents are ConcreteProducts. The Creater of our product is the application class. Concrete application classes as specified by the users of the framework equals ConcreteCreator in Figure 5.3.

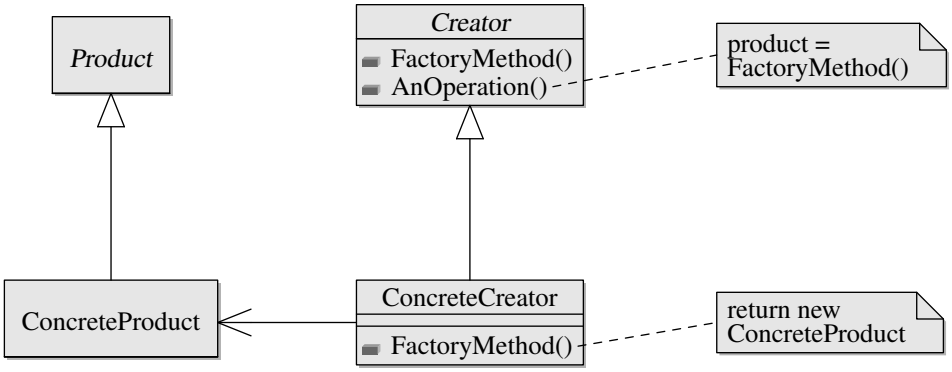

Figure 5.3: Factory Method pattern UML

Analysis

Creating a subclass of a Creator just to create a new product, and not because the creator must be subclassed anyhow, lead to a new point of evolution in the design that must be dealt with [9, p. 109].

Parameterization of the factory method can help keep the need for subclassing to a minimum. Depending on the implementation this might lack the safe static properties of the subclassing technique.

What the factory method essentially does is to remove a dependence between an exact product class and the creator class, thereby letting the product class created vary.

Abstract types might let us express this, the following shows an attempt.

```
trait Document {
  def open
  def close
}

trait Application {
  type D <: Document
  var docs = List[D]()

  def newDocument = {
    val doc = new D  // Illegal!
    docs = doc::docs
    doc.open
  }
}
```

The trait Document is our abstract product while Application plays the role of our creator. The abstract type member **type** D <: Document clearly states that our application trait depends on some subtype of Document. The method newDocument tries to create a **new** D. The Scala compiler will refuse to do this with the error message "class type required but Application.this.D found". There are several reasons this cannot be done, e.g. the exact type might be another trait that extends Document, traits cannot be instantiated. In Scala we have no way of constraining the type to be some class type with an appropriate constructor.

As in the original solution we are forced to use a factory method.

Componentization There is very little boilerplate in the Scala solution, so not much is gained by modularizing. We could try to write a reusable trait with a higher-order creation method. When using the trait, one would pass a creation method, along with the needed arguments. Creation methods needs a variable number of arguments, with differing types, but we cannot abstract over this. Any abstract creation method would have a fixed number of arguments, which severely limits is usefulness. The conclusion is that componentization is not relevant, because we lack the needed abstraction mechanisms.

Scala solution

The following solution uses a factory method.

Listing 5.3: Factory Method solution

```
trait Document {
  def open
  def close
}
```

```
trait Application {
  type D <: Document
  var docs = List[D]()

  def newDocument = {
    val doc = createDocument
    docs = doc::docs
    doc.open
  }
  //Factory method
  def createDocument:D
}
```

Using an abstract type gives us some advantages compared to the original solution. It opens up for more reuse of code in subclasses. It allows us to use generic collections and methods in the Application trait without losing type information in the subclasses of Application. The intent of the pattern in this case is arguably clearer, it is quite apparent that the Application class depends on some version of a Document. And we do not have to explicitly name a new subclass for each product, mixin composition provides a shortcut.

Summary Abstract type members removes the static dependency on a concrete product, and lets us write more reusable classes. Mixin composition and abstract type members let us avoid explicitly naming a new subclass for each product. This limits the inflexibility in using inheritance.

The presented solution has less implementation overhead than the original as soon as we need more subclasses of creator. Traceability is hardly an issue in the original solution, a naming convention for factory methods might be in place though. Regarding traceability in the new solution we could argue that it is improved, that is, if we agree that the intent is clearer when using abstract type members.

Summarising on the qualities compared to the original:

- Less implementation overhead: No need for explicitly creating a new subclass for each product, mixin composition provides a shorthand.
- More possibilities for reuse because of abstract type members.

5.4 Prototype

Description Intent of the pattern:

> "Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype." [9, p. 117]

Prototype presents an alternative to Factory Method (5.3) for decoupling of a creator class and its products. Instead of subclassing a given creator and overriding the factory method to return a specific class, composition is used instead. The creator is parameterized with a *prototype* object that provides a clone method for creating an instance of the object. Subclassing of the prototypical object, such as ConcretePrototype1 as seen in Figure 5.4, or parameterization, can be used for creating different objects. Parameterization helps reduce the number of classes in the system.

The pattern allows classes that are loaded dynamically to be instantiated at runtime. Typically the new prototype is added to an existing repository containing other prototypes [9, p. 119]. This allows the system to be very dynamic, we can compose new objects out of existing prototypes.

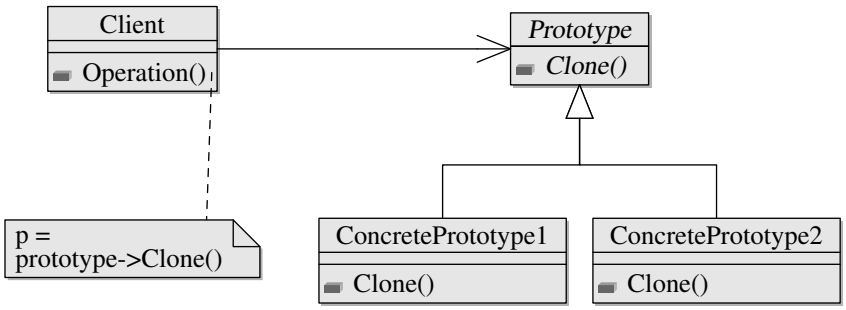

Figure 5.4: Prototype pattern UML

Analysis

Implementing the clone method can be non-trivial and touches upon the subjects of shallow and deep copy [9, p. 121]. Furthermore it might be impossible to add the clone method to existing classes, e.g. if we do not have access to the source, which might be the case in some scenarios.

Using *implicit conversions* [18][p. 15] we can add a clone method to an existing class. This opens up for the possibility of using the prototype pattern in new contexts. This can only be used in cases where the state needed for cloning an object is accessible through the public interface of the corresponding class.

As a side note, in prototype based languages such as Self, the pattern is supported by the language directly.

Componentization With the delicate problems of shallow and deep copy, it is hard to imagine a component that provides useful cloning functionality in general. Even though Scala does provide a `clone()` method in its standard class hierarchy, it will most likely have to be overridden to be useful.

Scala solution

A simple example of adding a clone method to an existing class is shown in Listing 5.4. The new clone method, in this case named `copy` is located in the class `CloneA`. When the copy method is invoked on `a`, the compiler will insert a call to the implicit method `defcloneA(a:A):CloneA` that returns a new `CloneA` object, the clone method is then invoked on this object resulting in a copy of the original `a`.

Listing 5.4: Adding a clone method

```
class CloneA(a:A) {
  def copy = new A(a.state)
}

implicit def cloneA(a:A):CloneA = new CloneA(a)
```

```
class A(var state:Int)

def main(args:Array[String]) = {
  def a = new A(2)
  println(a.state)
  def aCopy = a.copy
  println(a.state)
}
___
2
2
```

Summary Scalas implicits allows the pattern to be used in new contexts. Scalas features provides no special support for the pattern otherwise. Prototype based languages have direct support for the pattern. Summarising:

- The Scala solution enables the pattern to be used in new contexts.

5.5 Singleton

Description Singletons intent is:

> "Ensure a class only has one instance, and provide a global point of access to it." [9][p. 127]

Ensuring a class only has one instance can be paramount in different scenarios. E.g. it might be necessary for an underlying file system to only be represented by a single instance in a program, in order to avoid file corruption caused by simultaneous writes to the same file.

As seen in Figure 5.5, the singleton is typically implemented with a private constructor, a static field that stores the single instance and a static method for retrieving the instance. The static method Instance() serves as the global point of access. In the logic of Instance(), we can control the creation of the class.

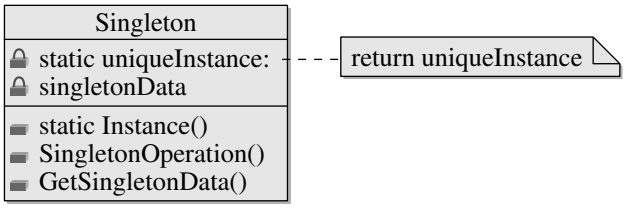

Figure 5.5: Abstract factory pattern UML

The solution above has a number of consequences. It is an improvement compared to global variables, since we avoid polluting the namespace with sole instances. It permits a variable number of instances to be created, all we have to do is to change the implementation of Instance(). Since the pattern is implemented with a class, we can refine it, letting our application configure the instance as needed [9][p. 128].

Analysis

Scala allows one to create singleton objects using the keyword **object**. A singleton object cannot, and need not, be instantiated with **new**. It is essentially automatically instantiated the first time it is used, and as the "singleton" in its name implies, there is ever only one instance.

Using a singleton object to implement the pattern partly captures the patterns intent. But the usage interferes with some of consequences discussed earlier. We loose the ability to extend the singleton class and we loose the ability to vary the number of instances. A singleton object can extend other classes (which allows us to factor out commonalities between different singleton objects), but we cannot extend an existing singleton object.

Companion objects are needed if we want the same consequences as the original pattern implementation.

Componentization The pattern is partly already present in the language, so will only discuss the case where refinement of the singleton is possible. In writing a reusable singleton we are faced with the problem of *hiding* a constructor that we do not have control over. This is not possible in Scala and would require a meta-object protocol in the language. The user of the component must necessarily provide us with a class that should function as a singleton, but we have no way of modifying existing constructors in this class.

Scala Solution

The following singleton implementation uses a companion object. This implementation is only relevant if we need to be able to refine the singleton. Otherwise the succinct solution using a singleton object is preferable.

Listing 5.5: Singleton in Scala

```
// companion object
object Singleton{
  private val instance:Single = new Single
  def getInstance() = instance
}

class Singleton private() // private constructor

val s = Singleton.getInstance()
```

Summary The presence of singleton objects allows to implement a straight-forward version of the pattern. If the ability to extend the singleton is important, an alternative solution using companion objects exists. This solution has the same characteristics as the original GOF solution. The solution using a singleton object, has the following qualities compared to the original:

- Succinct: Since we are using a built-in construct.

5.6 Summary

This chapter concludes our individual analysis of the creational patterns catalogued by [9].

Chapter 6

Structural patterns

This chapter will analyse all individual structural patterns (2.2) and if possible provide a reusable pattern in form of a component or an other otherwise improved implementation.

6.1 Adapter

Description Intent of the pattern:

> "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces." [9, p. 139]

Two basic types of adapters are presented by GOF, *class adapters* and *object adapters*. Each with different strengths and weaknesses.

A class adapter uses *inheritance* to adapt one interface to another. The Adapter class, shown in Figure 6.1, inherits from the Adaptee and the target interface represented by class Target. When a client invokes a specific operation on the target, the adapter class calls the appropriate inherited method from the adaptee class. Inheritance lets us refine the behaviour of the adaptee if needed, by overriding methods, but it also creates a static dependence on an exact adaptee class. This means that the adapter class cannot be reused with subclasses of adaptee [9, p. 142].

An object adapter uses composition/delegation instead of inheritance as the main mechanisms for realising the adaption. Figure 6.2 shows how, the Adapter class contains a reference to the Adaptee class. When a client invokes the Request method, the message is passed on to the adaptee object. Since the adapter just contains a reference to the adaptee, no static dependencies to a specific class is introduced, the adapter class can be reused with any subclass of Adaptee [9, p. 142]. This comes at the cost of introducing the self problem and it makes it impossible to override behaviour of the adaptee.

A *two-way adapter* provides transparency for all clients, those that expect the target interface *and* those that expect the adaptee interface.

Analysis

The 2nd principle of GOF is clearly in effect when considering the object adapter.

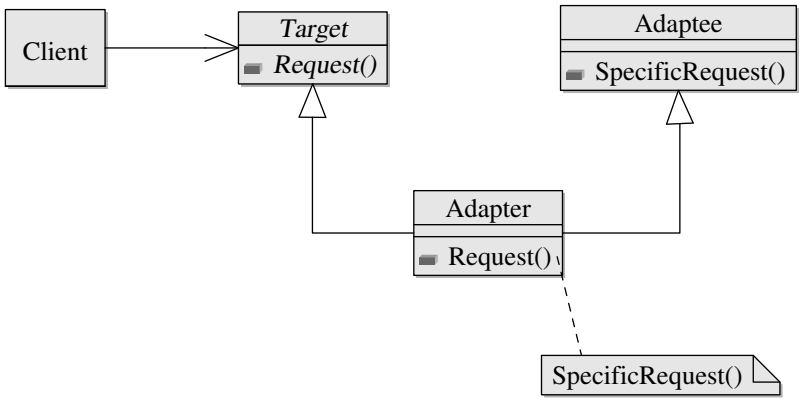

Figure 6.1: Class adapter pattern UML

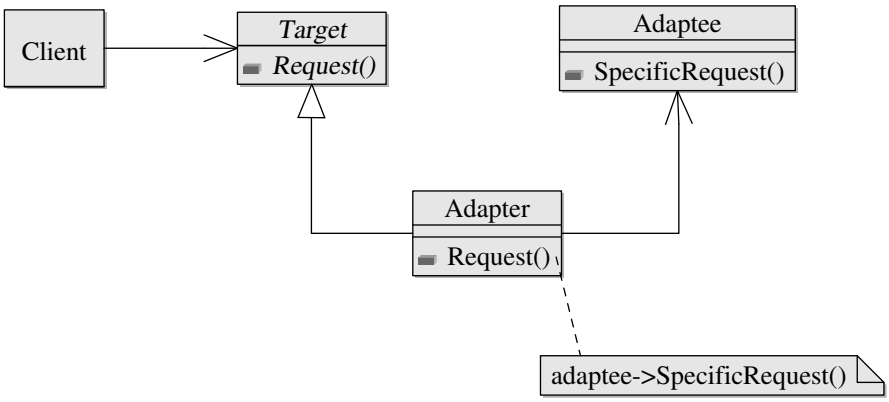

Figure 6.2: Object adapter pattern UML

An explicit self type can remove the dependence between adapter and a specific adaptee class which was a problem with the original inheritance based solution. An even more flexible solution, would be to use abstract members to declare the required methods, instead of a self type. Mixin composition lets us construct a two-way adapter, since the compound type of the result lets our adapter function both as target and adaptee via subsumption. Furthermore it removes the self problem.

The combination of the above features combines the strengths of the two GOF solutions. Except for one detail, which might be important depending on the scenario, we cannot change adaptee at runtime.

Componentization The things that varies from each instantiation are the types of adapter and adaptee and the communication between them. This communication is, amongst other things, characterized by the set of methods involved. In Scala there is no meaningful way to *abstract over the set of members in a class*, this implies that we cannot provide a useful component.

The adapter is reusable with any subclass of adaptee though, but again, this pertains only to a specific context in which the pattern has been applied.

Scala solution

The Scala solution combines most of the benefits of a class adapter and object adapter in one solution. At the same time it also functions as a two-way adapter. Listing 6.1 shows a simple example. The key to the solution lies in the trait Adapter. Its explicit self type, the compound type Target **with** Adaptee, removes the dependency on a specific Adaptee class inherent in the class adapter solution from GOF.

Listing 6.1: Adapter in Scala

```
trait Target {
  def f
}

class Adaptee {
  def g = println("g")
}

trait Adapter {
  self: Target with Adaptee =>
  def f = g
}

def main(args:Array[String]) = {
  val adapter = new Adaptee with Adapter with Target
  adapter.f
  adapter.g
  ()
}
___
g
g
```

The main method shows how we can create an adapter. Invoking the method f and g illustrates that we are also dealing with a two-way adapter.

Summary The Scala solution combines almost all of the benefits of class and object adapters and removes the self problem. The usage of a self type lifts the dependence on a specific adaptee class. An alternative to this is to declare abstract members in the Adapter class, this gives us even more flexibility, since only structural requirements are specified. The solution gives us some of the flexibility that the composition based solution has, which suggests that the 2nd principle might be weakened when applied to Scala. No component was provided, since we cannot abstract over the member set in a class.

Compared to the original GOF solutions, our solution has the following qualities:

- No self problem, but almost as flexible as an object adapter.
- No dependency on a specific adaptee class, as compared to the original class adapter.
- Only applicable if we do not need to change adaptee dynamically.

6.2 Bridge

Description Intent of the pattern:

> "Decouple an abstraction from its implementation so that the two can vary."
> [9, p. 151]

Using the bridge pattern we can avoid a permanent binding between an abstraction and its implementation. The bridge pattern is a relevant design when one has a proliferation of classes, typically from a class hierarchy defining some central abstractions, refined by inheritance, that each need different implementations. A motivating example involves a window abstraction in a user interface framework [9, p. 151-52]. The central abstraction, the Window, is refined in subclasses. Each of these refined subclasses is then subclassed with appropriate implementations for each supported platform, which leads to the proliferation. This creates a permanent binding between abstraction and implementation and makes the system rigid. Change in one abstraction leads to changes in all subclasses. The solution is to split the hierarchy in two independent hierarchies. One for the abstraction and one for the implementation. This reduces the amount of classes in the system and lets each hierarchy evolve independently. Figure 6.3 shows the relationship between the two hierarchies.

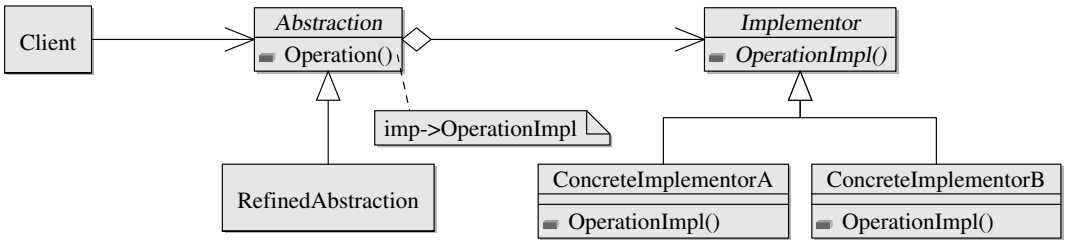

Figure 6.3: Bridge pattern UML

All operations in Abstraction is implemented with the abstract methods located in Implementor . This is the key to the decoupling.

Analysis

The bridge itself is realised with composition and delegation. The abstraction contains a reference to an implementor. The exact type of the implementor is hidden with an abstract class, which combined with the composition enables the decoupling. GOFs second principle is in play here.

Instead of enabling the refactoring of the class hierarchy with composition and delegation, we can in Scala use explicit self types. This comes with the cost that we cannot change the binding at runtime.

Componentization As in Adapter (6.1) we are faced with the impossibility of abstracting over communication between classes in a meaningful way. The communication between the abstraction and implementor hierarchy will differ wildly between different instantiations of the pattern. The only things that stay the same between different instantiations are the different roles involved, which is on a conceptual level, this implies that there is nothing concrete to modularize.

Scala solution

In Listing 6.2 the root of the abstraction hierarchy, named Window, has a self type that refers to the root of the implementor hierarchy, WindowImp.

Listing 6.2: Bridge in Scala

```
trait Window {
  self: WindowImp =>

  def drawRect(x1:Int,x2:Int,x3:Int,x4:Int) = {
    drawline(x1,x2)
    drawLine(x1,x3)
    drawLine(x2,x4)
    drawLine(x3,x4)
  }
}

// abstractions
trait TransientWindow {
    self: Window =>
    def drawCloseBox = drawRect(4,3,2,1)
  }
trait IconWindow {
    self: Window =>
    def drawBorder = drawRect(1,2,3,4)
}

// common interface for all implementors
trait WindowImp {
  def drawLine(x:Int,y:Int)
}

// implementors
trait WindowOSX extends WindowImp  {
  def drawLine(x:Int,y:Int) = println("drawing line in OSX")
}

trait WindowVista extends WindowImp {
  def drawLine(x:Int,y:Int) = println("drawing line in Vista")
}

def main(args:Array[String]) = {
  val windowOSX:Window = new Window with WindowOSX
  windowOSX.drawRect(1,2,3,4)
}
___
drawing line in OSX
drawing line in OSX
drawing line in OSX
drawing line in OSX
```

As mentioned in Section 3.5.1 we are forced to repeat the self type annotation in subclasses of Window. An alternative to using a self type annotation, although more verbose, is to define the needed operations as abstract members. This is even more flexible as we do not name specific traits.

Summary Self types, or abstract members, lets us express the dependence between the two hierarchies. The self problem no longer exist with the new solution, since we do not use composition/delegation anymore. What removed the problem was the usage of mixin composition. If we accept this version of the pattern, the 2nd principle is under attack again. The presented solution is only applicable if we do not need to change binding at runtime.

No component was provided, since it was deemed impossible to abstract over communication between abstraction and implementor hierarchy.

Summarising on the qualities compared to the original GOF solution:

- No self problem because of mixin composition.
- Solution only applicable if we do not need to switch bindings dynamically.

6.3 Composite

Description The intent of the pattern is:

> "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." [9, p. 163]

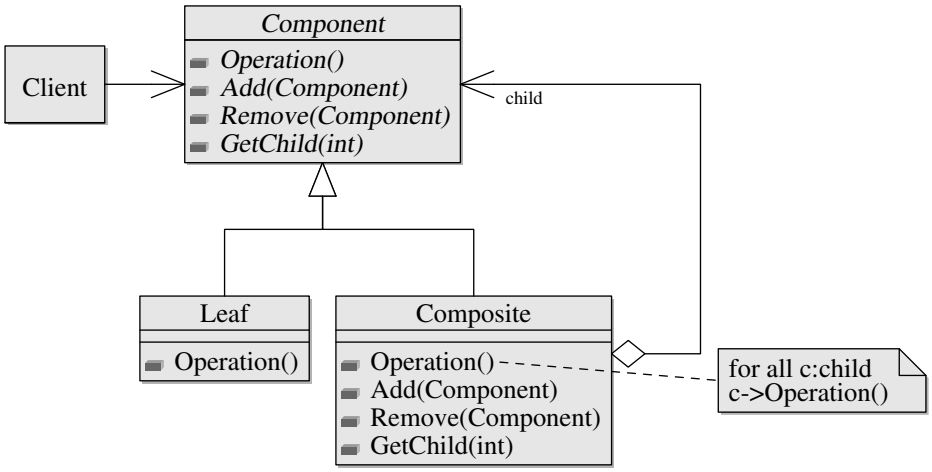

Figure 6.4: Composite pattern UML

A domain where the composite pattern can be applied naturally is in the context of a text editor. Documents consists of pages which consists of columns, which in turn consists of lines of text and images, and so forth. Often we need to perform the same operation on either a part of such a tree structure or a leaf, maybe deleting a whole column at once, or just a single character. In order to support this, an interface specifying the operations needed, are implemented by all leaf objects *and* all *composite* objects.

Often the composite structure is traversed in some way. Optional parent pointers in leafs or composites helps the traversal algorithm.

The GOF solution can be seen in Figure 6.4. Class Component specifies the common interface among leafs and composites. The methods Add,Remove and GetChild are also part

of the Component interface. This is done for the sake of *transparency*, leafs and composites should be handled the same way by clients.

Analysis

The common interface shared by leafs and composites causes some problems. Since leafs cannot contain children, adding a child to a leaf via the Add method should raise an exception and this must be handled in client code. The other child operations should be handled analogously.

It could be argued that there are conceptual issues with the GOF solution, since defined methods in a class should make sense in the context of the class. This gives rise to another version of the pattern which according focuses on *safety* [9, p. 167]. In this version the child operations are only part of the composite interface. Any attempt to add a component to a leaf, will now be caught at compile time. The drawback is that clients, depending on the type information available, must explicitly distinguish between components. A Boolean isComposite() method is typically made available for all components. Afterwards a type cast is needed, which again raises some safety issues. Another solution would be to add a GetComposite() method [9, p. 168]. In the case of a component it would return a null value and in the case of a composite it would return a reference to self.

Independent of which of the above solutions we prefer we *cannot escape the fact that composites and leaf objects essentially are different.* Depending on the solution, this difference must be dealt with in certain situations.

Tree structures are often modelled with algebraic data types in functional languages. Scalas variant, case class hierarchies, offers a different way of implementing composite structures as opposed to the GOF solution, which inherently deals with the above mentioned issue.

Componentization There is very little boilerplate code that needs to be written in order to implement the pattern in Scala. The handling of the child elements in the original solution has potential for modularization, but this is not present in the Scala solution. What we could try to abstract over is the possible operations that can be performed on the composite structure. This would involve creating traits with methods with generic type signatures, that then would have to be refined in subclasses etc. We would end up writing more code than what is needed for writing a specific interface trait for the composite operations. We would also loose the meaningful names that our tailor made trait would benefit from, and we still cannot abstract over a set of operations.

Scala solution

In Listing 6.3 the abstract class Component represents the interface for the common operations we want to be able to perform on leafs or composites, in this case only one method named display. The case classes Text and Picture are two leaf objects, they both implement display. The case class Composite contains any number of child components, that is, leafs or other composites. Note that the children variable is mutable, i.e. we can change the tree structure without creating a new tree. Depending on how heavyweight our leaf objects are, or whether they serve other roles in our design or not, we could make the children variable immutable as well as mutable.

Listing 6.3: Composite implementation in Scala

```
sealed abstract class Component {
  def display
}

case class Text(var text:String) extends Component {
  def display = println(text)
}

case class Picture(var picture:String) extends Component {
  def display = println(picture)
}

case class Composite(var children:List[Component]) extends Component {
  def display = children.foreach( x => x.display )
}

def main(args:Array[String]) = {
  val tree =
  Composite(List(Composite(List(Text("t1"),Picture("p1"))),Text("t2")))
  tree.display
  tree.children(1).display
}
---
Text t1
Picture p1
Text t2
```

The main method shows an example construction of a composite. We call the display method on the whole composite structure and a specific leaf object to illustrate that we can deal with them in a uniform way.

We take advantage of the implicit factory methods present in our case classes, but this is not the main reason for adding the **case** keyword to our classes. The reason is that this gives us a type-safe way of traversing and altering the structure via pattern matching. With pattern matching we do not need to explicitly check whether a given component is indeed a composite, e.g. when adding or removing children, we also avoid the associated type casts. If we have a match, we know it is a composite. This removes the need to have children handling methods in leaf objects, thereby removing the conceptual issues involved. It also removes the need of any isComposite method if the implementation strategy of not having uniform interfaces was taken instead.

The following shows an example of a method that traverses a composite structure and changes all Text nodes in-place. The @ operator binds the left-side variable to the matching pattern.

```
def changeAllText(c:Component, s:String):Component = c match {

  case tx @ Text(v) => {
    tx.text = s
    tx
  }
  case p @ Picture(v) => p
  case c @ Composite(cs) => { c.children =
    for {
```

```
        c <- cs
        cnew = changeAllText(c,s)
      } yield cnew
      c
    }
  }
```

This method can off course be placed anywhere, including the composite structure, which goes to show that we are still dealing with object-oriented constructs and not standard algebraic data types. Note that the structure is changed in-place, which is atypical in functional languages, this is to preserve any state our nodes might have that is not visible to us in the match.

Summary Scalas case classes offers a convenient way of composing and decomposing composite structures. The object-oriented version of algebraic datatypes ensures that other design patterns can be applied to any classes or objects involved in the structure.

The implementation overhead in Scala for implementing the pattern is minimal, the pattern is integral to case classes and decomposition of the created hierarchy with pattern matching.

Summarising:

- No boilerplate code, e.g. handling of child elements.
- Type safe and explicit handling of distinction between composite and leaf objects.

6.4 Decorator

Description Intent of the Decorator pattern:

> "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." [9, p. 175]

A typical example of applying the pattern to a concrete design, is in the domain of graphical user interfaces [9, p. 175]. A GUI window often need scrollbars, either vertical, horizontal or both, depending on the runtime context. One way of handling this is by extending the window class at compile time through inheritance, e.g. WindowWithVerticalScroll or perhaps WindowVerticalAndHorizontalScroll. As the long names might suggest this could lead to an exponential growth of the class hierarchy, fueled by the act of adding different features or responsibilities to the *class* instead of what we really need; adding the responsibility to a specific *object* at runtime. The pattern allows deferring the decision of inclusion or exclusion of specific responsibilities to runtime, thereby dynamically composing the features.

Analysis

GOFs second principle is clearly in play here, the dynamic nature of composition enables the solution.

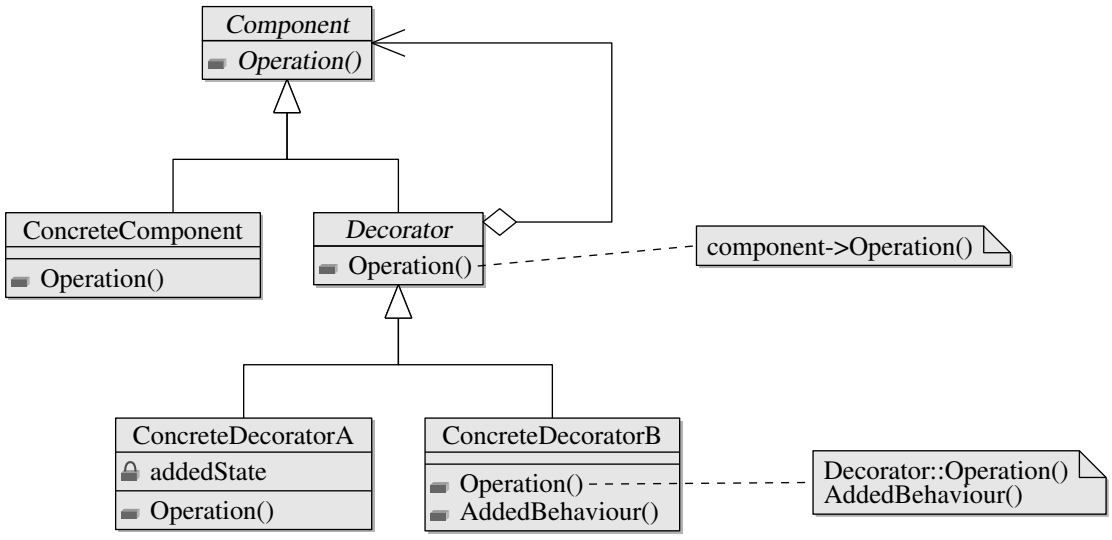

Figure 6.5: Decorator pattern UML

A design that uses Decorator often results in a system composed of lots of little objects that all look alike. The objects differ only in the way that they are interconnected, not in their class. This makes the system easy to customize but hard to learn and debug [9, p. 178].

Another problem with the solution is: "A decorator and its component are not identical". The pitfall here is not to rely on object identity in ones code [9, p.178].

The use of composition is the root of these problems, using mixin composition instead would avoid the issues. Mixin composition is essentially just a shorthand for defining anonymous classes (3.2) and it is a compile time construct. The different combinations needed would have to be defined a priori at compile time and we would not be able to change the decorations at runtime.

Using the technique of stackable modifications mentioned in Section 3.2.2, we can implement a decoration.

```
class TextView(var s:String)  {
  def draw  = println("Drawing.." + s)
}

trait BorderDecorator extends TextView {
  abstract override def draw = { super.draw ; drawBorder }
  def drawBorder = println("Drawing␣border")
}

val textViewWithBorder = new TextView with BorderDecorator
textViewWithBorder.draw
```

The value textViewWithBorder is one such definition, but it is not possible to switch the trait BorderDecorator into another decoration at runtime.

If mixin composition was a dynamic construct instead, enabling us to compose traits by name at runtime and at the same time allowed us to dynamically change the traits in

the composition, we would have an ideal solution. But this is essentially *dynamic inheritance* which is not present in Scala.

If we encapsulate our TextView we could reuse it in different compositions thereby allowing us to change decorations. This leads to a solution like this.

```
trait Component {
  def draw
}

class EncapsulateTextView(c:TextView) extends Component {
  def draw = c.draw
}

class TextView(var s:String) extends Component {
  def draw  = println("Drawing.." + s)
}

trait BorderDecorator extends Component {
  abstract override def draw = { super.draw ; drawBorder }
  def drawBorder = println("Drawing border")
}
trait ScrollDecorator extends Component {
  abstract override def draw = { scrollTo ; super.draw }
  def scrollTo = println("Scrolling..")
}

 def main(args: Array[String]) = {
  val tw = new TextView("foo")
  val etw1 = new EncapsulateTextView(tw) with BorderDecorator with ScrollDecorator
  etw1.draw
  tw.s = "bar"
  val etw2 = new EncapsulateTextView(tw) with ScrollDecorator with BorderDecorator
  etw2.draw
  ()
}

---

Scrolling..
Drawing..foo
Drawing border
New decorators
Scrolling..
Drawing..bar
Drawing border
```

Each different combination would still have to be defined a priori and we do not get rid of the problem of identity. For these reasons the original GOF solution using composition is preferred, since changing decoration at runtime is essential in the patterns intent.

Componentization The only thing that stays the same between different instantiations is the reference from one decoration to the next, leaving very little room for modularization according to the less tokens expended rule.

Summary Scala does not offer something new with respect to the Decorator pattern. A solution was rejected, since the ability to dynamically change decoration is essential in the pattern, this is clearly stated in the intent. No component was provided.

6.5 Facade

Description Intent of the pattern:

> "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." [9, p. 185]

Facade provides a simple interface to a complex subsystem [9, p. 186]. This is done in order to shield clients, that does not need the lower level functionality provided by subsystems, from the complexity. Clients communicate with the subsystem by sending requests to a facade, the facade translates and handles intercommunication between the subsystems [9, p. 187].

Facades can be used to layer a system by decoupling subsystems. A specific facade functions as an entry point to the subsystems it encapsulates [9, p. 186].

The use of facades as entry points promotes weak coupling between subsystem components and lets you vary the components without affecting the clients. [9, p. 186]. Making subsystem classes private would be useful, but few object-oriented languages support it [9, p. 188].

Facade objects are often Singletons [9, p. 193].

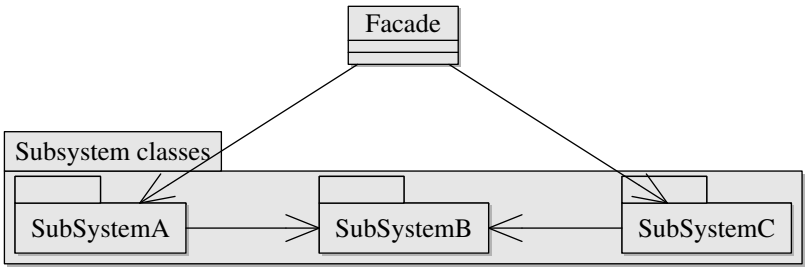

Figure 6.6: Facade pattern UML

Analysis

The term "subsystem" is used in the intent deliberately. We are talking at a higher level than traditional classes. Since Scala modules are first-class, combined with nesting, access modifiers and fine-grained import, we have great flexibility in implementing the pattern. Top level classes can be thought of as facades themselves, with nested classes as submodules. Abstract types lets us define a facade that is reusable in a certain application context, since we can remove the static dependencies on specific subsystem classes. Abstract members lets us specify the required submodules that the Facade depends on.

Regarding facades as entry points, they themselves can be nested as well in top level traits, again using abstract types and members to remove static dependencies. The constructs involved scales.

Componentization The things that differ are the subsystems, their intercommunication, which we cannot abstract over, and the interface of the facade itself. Nothing stays the same between different instantiations, which means there is nothing to modularize and reuse.

Scala Solution

The trait Facade from Listing 6.4 is an abstract facade with nested subsystem classes that are only accessible by Facade and subclasses.

The method foo is part of the public interface of the facade accessible by clients, it calls one of the subsystems with another subsystem as argument.

The singleton object FacadeA is an example of a concrete facade.

Listing 6.4: Facade in Scala

```
trait Facade {
  type A <: SubSystemA
  type B <: SubSystemB

  protected val subA:A
  protected val subB:B

  def foo = subB.foo(subA)

  protected class SubSystemA
  protected class SubSystemB {
    def foo(sub:SubSystemA) = println("Calling␣foo")
  }
}

object FacadeA extends Facade {
  type A = SubSystemA
  type B = SubSystemB
  val subA:A = new SubSystemA
  val subB:B = new SubSystemB
}
```

Summary Top-level classes can be seen as facades with subsystems as nested classes. Access modifiers allows us to hide the subsystem classes if needed. No component was provided, since Scala did not have the necessary abstraction mechanisms needed and no boilerplate code was present either.

Summarising on the qualities of our solution compared to the original:

- Subsystem classes are hidden from clients.
- More oppurtinies for reuse and refinements of facades, because of abstract types.

6.6 Flyweight

Description Intent of the pattern:

> "Use sharing to support large numbers of fine-grained objects efficiently." [9, p. 195]

The motivating example involves an object-oriented document editor. Objects are used to represent embedded elements like tables and figures. Representing individual characters with objects would enable a uniform way of drawing and formatting all elements, but doing so would be very expensive at runtime. [9, p. 195].

The key idea is to separate an objects state into *extrinsic* and *intrinsic* state. The intrinsic state can be shared on a wide scale, minimizing storage requirements, while the extrinsic state can be computed on the fly, trading computation for storage. Intrinsic state in our example would be characters in the alphabet, while extrinsic state could be the position in the document. The extrinsic state is needed when drawing is performed.

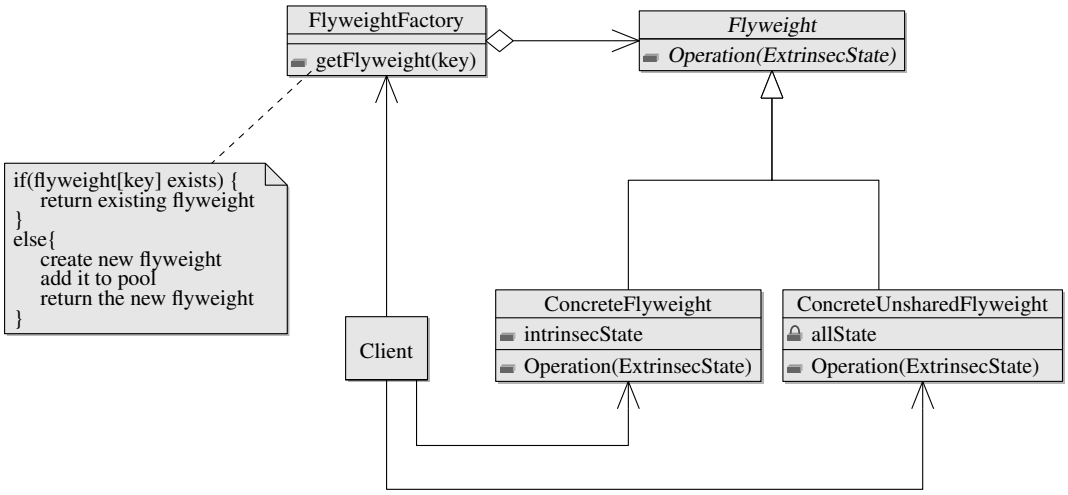

Figure 6.7: Flyweight pattern UML

As seen in Figure 6.7, a *FlyweightFactory* handles a pool of flyweights. When requested for a specific flyweight it will return a shared one from the pool or create a new one and return it, if it is not present the pool yet.

Analysis

Scala provides no specific constructs for dealing with Flyweight objects themselves, but we can write a reusable Flyweight factory, as is discussed next.

Componentization The things that differ between each instantiation are the concrete types of the flyweights. The flyweight factory is a part of each instantiation. If enough boilerplate code is present in a factory there might be room for componentization.

The flyweight pool is essentially an associative collection from intrinsic state to a specific flyweight object. By using a type parametrized trait we can abstract over this. With the unification of functions and classes we can implement a flyweight factory with some convenient syntactic shortcuts. Our reusable flyweight factory in Listing 6.5 is inspired by the implementation of the Array[T] class mentioned in Section 3.3.1.

Listing 6.5: Flyweight Factory Component

```
trait FlyWeightFactory[T1,T2] extends Function[T1,T2]{
  private var pool = Map[T1,T2]()
  def createFlyWeight(intrinsic:T1):T2

  def apply(index:T1):T2 = {
    pool.get(index) match {
      case Some(f) => f
      case None =>  {
        pool += (index -> createFlyWeight(index))
        pool(index)
      }
    }
  }
  def update(index:T1, elem:T2) { pool(index) = elem }
}
```

The justification of the component is the logic in the apply method and the clever extension of the Function[T1,T2] trait.

Scala Solution

This section illustrate the usage of the flyweight factory. The Character class is our flyweight, it depends on a DrawingContext class provided by clients. The context provides extrinsic state that is needed in our drawing method. The singleton object CharacterFactory implements the factory method createFlyWeight(c:Char), which is all that is needed in order to use the factory.

Listing 6.6: Example usage of reusable flyweight factory

```
trait DrawingContext { def queryExtrinsicState }

class Character(val char:Char) {
  def draw(context:DrawingContext) = println("drawing character")
}

object CharacterFactory extends FlyWeightFactory[Char,Character] {
  def createFlyWeight(c:Char) = new Character(c)
}

val f1 = CharacterFactory('a')
val f2 = CharacterFactory('b')
val f3 = CharacterFactory('a')
```

Note that f1 and f2 points to the same shared flyweight object.

Summary The existence of functional abstraction lets us write a reusable Flyweight Factory component. Even though the reusable Flyweight Factory does not contain much logic, the less tokens rule is not applicable, partly because of the technique of implementing the factory as a function allowing for concise syntax in the usage.

Summarising:

- Reusable flyweight factory component.

- Improved traceability because of the use of a component.

6.7 Proxy

Description The patterns intent is

> "Provide a surrogate or placeholder for another object to control access to it."
> [9, p. 207]

GOF defines several types of proxies, "A virtual proxy creates expensive objects on demand" [9, p. 208] the rationale is "One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it".

A remote proxy "provides a local representative for an object in a different address space" [9, p. 208].

A protection proxy controls access to an object, thereby enforcing some security policy.

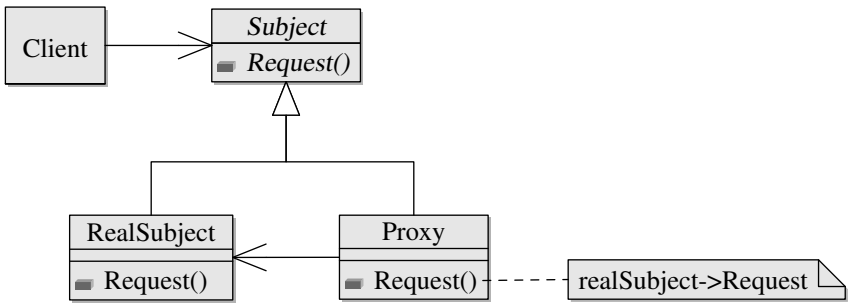

Figure 6.8: Proxy pattern UML

Proxy contains a reference to the RealSubject and handles all incoming requests from clients, it is this level of indirection that is the key to the pattern. They both share the same interface Subject, since clients should not be able to distinguish them.

Analysis

Because of the use of delegation, the pattern suffers from the self problem.

Using the **lazy** keyword we can defer the full cost of initialization of any *value field*. This enables some aspects of virtual proxies to be implemented easily without implementing the full pattern and it eliminates the self problem in these cases.

Scala does not provide any new constructs for an improved implementation otherwise.

Componentization What differs from each instantiation are the types of Subject, RealSubject and Proxy. Only the fact that a Proxy must contain a reference to a RealSubject stays the same, but this is hardly worth componentizing and would not survive the less tokens expended rule.

Scala Solution

Using the keyword **lazy** dictates that a value is evaluated upon demand.

```
class VirtualProxy {
  lazy val expensiveOperation = List(1 to 1000000000)
}
```

Summary One of Scalas features is lazy evaluation, this gives direct language support for some aspects of the Proxy pattern. Very little boilerplate code exist in the pattern, making it a bad candidate for componentization.

Summarising:

- Trivial implementation of certain 'Virtual proxies'.
- No implementation overhead: Since we have no message forwarding from a proxy.

6.8 Summary

This chapter concludes our individual analysis of the structural patterns cataloged by [9].

Chapter 7

Behavioral Patterns

This chapter performs an individual analysis of all behavioral patterns (2.2) cataloged by GOF. If possible a component or an other otherwise improved implementation is provided.

7.1 Chain of Responsibility

Description Intent of the pattern:

> "Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it." [9, p. 223]

The idea of the pattern is to decouple senders and receivers of a message. The pattern lets the exact receiver of a message be determined at runtime.

Figure 7.1 shows the basic UML for the pattern. The concrete handlers are in our case the different widgets. Method HandleRequest() defines the criteria for accepting a message. Any Handler object has a reference to a successor object in the chain, the reference is used for passing on the message if it is not handled.

The common interface Handler ensures that the specific receiver remains unknown.

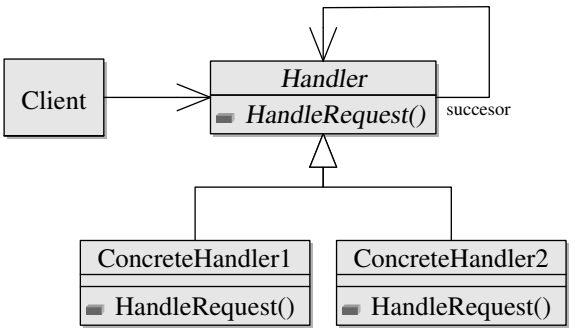

Figure 7.1: Chain of responsibility pattern UML

The solution makes it possible to create the chain dynamically, by adding or removing objects, thereby changing responsibilities for handling a request, while maintaining low coupling between the objects involved.

The HandleRequest method can optionally be parameterized with arguments letting the callees criteria for handling the request be based on the value of the arguments.

Analysis

If there are no preexisting references for defining a chain, boilerplate code for handling successor links must be rewritten each time the pattern is implemented. This pattern specific code must be located in each object in the chain, reducing locality of pattern specific code.

Componentization In Scala we can avoid writing boilerplate code for handling the successor link and for propagation of the request. This logic stays the same between different instantiations. Listing 7.1 shows a reusable handler. The generic trait Handler[T] abstracts away the type of the requests propagated through the chain. A participant of the chain must mixin the handler trait and overwrite the criteria for handling a specific request, that is, method handlingCriteria(request:T):Boolean. Furthermore, the participant must overwrite the method doThis(v:T):Unit which is called if the criteria is met, otherwise the request will propagate further down the chain. The default implementations present will automatically reject a request and send it to the next link.

Listing 7.1: Reusable generic handler trait

```
trait Handler[T] {
 var successor:Handler[T] = null

  def handleRequest(r:T):Unit =
  if (handlingCriteria(r)) doThis(r)
   else if (successor != null)
    successor.handleRequest(r)

  def doThis(v:T):Unit = ()
  def handlingCriteria(request:T):Boolean = false
}
```

Scala solution

The following listing will illustrate how the handler trait can be used. The class Sensor forms the start of a chain consisting of Handler[Int] objects. Classes Display1 and Display2 handles the sensors readings, depending on the size of the output. Notice the alternative implementation of Display2, *all pattern specific code* pertaining to the class is kept in a separate trait Display2Handler.

Listing 7.2: Example usage of handler trait

```
class Sensor extends Handler[Int]  {
  var value = 0
  def changeValue(v:Int) {
    value = v
    handleRequest(value)
  }
}
```

```
class Display1 extends Handler[Int] {
  def show(v:Int) = println(v)
  override def doThis(v:Int) = show(v)
  override def handlingCriteria(v:Int):Boolean = v < 4
}

class Display2  {
  def show(v:Int) = println(v)
}

 // another solution, pattern specific code is kept in separate trait
 trait Display2Handler extends Display2 with Handler[Int] {
   override def doThis(v:Int) = show(v)
   override def handlingCriteria(v:Int):Boolean = v >= 4
 }

 def main(args: Array[String]) = {
   val sensor = new Sensor
   val display1 = new Display1
   val display2 = new Display2 with Display2Handler
   sensor.successor = display1
   display1.successor = display2
   sensor.changeValue(2)
   sensor.changeValue(4)
   ---
   Display 1 2
   Display 2 4
```

A problem with any solution using the generic handler from Listing 7.1 is that we can inherit only once from the handler trait, which means that objects cannot participate in several chains.

Summary The generic handler traits solves the problem of rewriting the code for propagation of requests and handling of successor link. It also makes it quite apparent that a given class is part of the pattern.

The chain of responsibility pattern is often used in conjunction with the Composite pattern [9]. Parent references in a component can be used as successor links, thereby using the structure that is already present. Our Composite solution (6.3) does not use parent references in the composite structure which are used for traversals, instead pattern matching is applied on the structure for any traversal code needed. We are free to create any chain in the hierarchy, not just one based on the existing structure, since we can arbitrarily mix in the handler trait and create references between the links.

Summarising:

- Pattern is componentized, resulting in improved reusability and
- Improved traceability.
- Pattern specific code can be completely localised, which makes it non-intrusive and improves traceability.

7.2 Command

The intent of the command pattern is:

> "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo-able operations." [9, p. 233]

Typical GUI frameworks contains widget classes such as buttons or menu items that needs to execute some request on behalf of the user when clicked or selected. The framework designer has no way of knowing in advance exactly which class or object that will be the receiver of the request [9, p. 233]. The command pattern provides a way of maintaining a *binding between a receiver and a request* [9, p. 236].

Commands are an object-oriented replacement for callbacks known from procedural languages [9, p. 245]

Since operations are encapsulated in objects and can be queued or stored, the pattern also supports macros. E.g., we could record a number of requests and store the sequence for later execution.

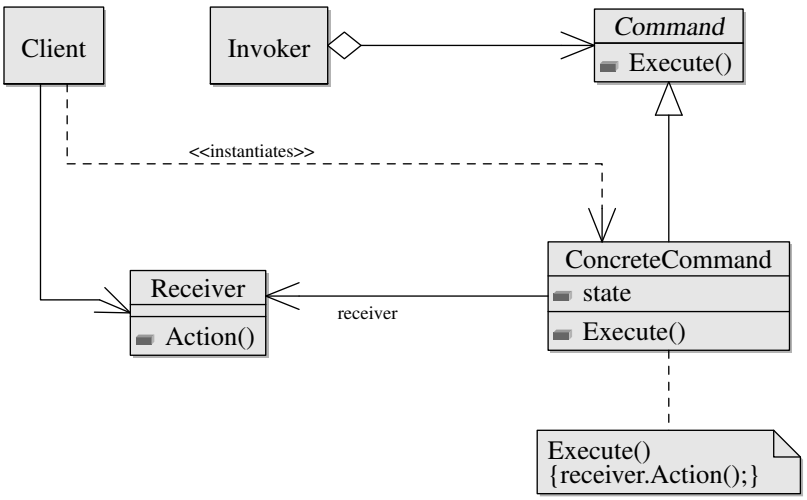

Figure 7.2: Command pattern UML

Analysis

Callback functions are easily implemented in Scala because of first-class functions. They gives us great flexibility in defining a binding between a receiver and a request. Macros can be implemented with a collection that stores first-class functions. If we want to support undoing of commands via history functionality, the bindings we can define gets more limited, since our first-class function must support an undo method which implies that it must be defined as an extension of a `Function` trait.

Logging of callbacks can be implemented with advice or directly in an `apply` method.

Componentization Callbacks are easily expressed in Scala, so there is not much to componentize. History functionality could be componentized but is not directly part of the pattern, the pattern enables it though.

Scala solution

The class Button expects a callback function that it will execute when the method click is called.

```
class Button(var click: (() => Unit))
val button = new Button(() => println("click!"))
button.click
---
click!
```

In order to support history functionality our command, or callback, we must support an undo method.

```
trait Undoable { def undo }

class CallBack extends (() => Unit) with Undoable {
  def apply():Unit = println("callback!")
  def undo = println("undoing!")
}

object History {
  var commands: List[(() => Unit) with Undoable] = List()
  val undoAll = commands.foreach(_.undo)
}
```

Summary The concept of callbacks known from procedural languages are directly expressible in Scala, enabling a straightforward implementation of the pattern. The unification of functions and classes gives us the expressivity to treat the callbacks as more than just pure functions enabling us to implement history functionality.

Summarising:

- Concise implementation: Callbacks are directly expressible.
- Arguably improved traceability: Language constructs matches pattern concept.

7.3 Interpreter

Description Intent of the pattern:

> "Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language." [9, p. 223]

As the intention states, the Interpreter pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences. [9, p. 243]. Given a grammar we create a class for each production rule, these are nodes in our abstract syntax tree (AST). Right-hand side symbols of each rule are instance variables of these classes. Each class contains an Interpret method that is responsible for the

actual interpretation of the node itself, the method typically accepts a Context object to access and store the state of the interpreter [9, p. 246].

The pattern is often used in combination with the Visitor and Composite patterns [9][p. 255].

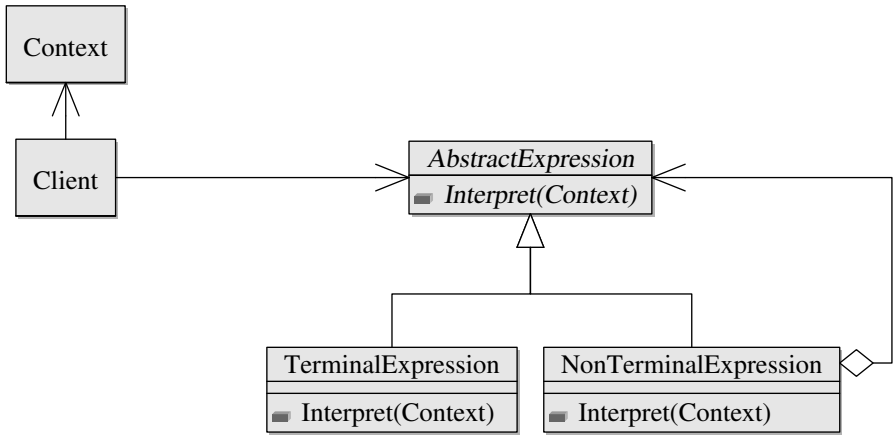

Figure 7.3: Interpreter pattern UML

Analysis

ASTs are often modelled with algebraic datatypes in functional languages. Scalas case classes are just as convenient for creating and manipulating abstract syntax trees. Instead of using the Visitor pattern as presented by GOF, we can use pattern matching as described in our Visitor (7.11) solution for defining operations on the tree.

Parser combinators are part of Scalas standard library [1]. The library provides an internal parser domain-specific language (DSL), allowing one to use EBNF like syntax to describe grammars. The output is typically an AST represented as a case classes hierarchy. This is outside the scope of the original pattern though, since the pattern does not deal with how to actually parse and create the AST. One could speculate that the pattern would be more popular if such libraries where available in mainstream languages.

Componentization Each instantiation is a different language which means that we need different case class hierarchies for each instantiation. In *general* there is nothing to modularize, but we could imagine families of related languages..

Scala Solution

The solution shown in Section 7.11 is essentially an application of Interpreter. The language is in this case a subset of arithmetic expressions, only summation expressions are present.

Summary The unification of algebraic datatypes and class hierarchies, combined with pattern matching and the presence of parser combinators in the standard library, makes Scala ideal for implementing the interpreter pattern.

[1] `http://www.scala-lang.org/docu/files/api/scala/util/parsing/combinator/Parsers.Parser.html`

Summarising:

- Concise and straightforward implementation of ASTs with case classes.
- Pattern matching is a good fit for manipulation of ASTs.
- Parser combinator present in standard library.

7.4 Iterator

Description The intent of the Iterator pattern:

> "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." [9, p. 257]

The basic idea in the pattern is to remove the responsibility of *traversal* and *access* from the aggregate object, such as a list, to an *iterator object*. This enables different traversal strategies to be handled in a uniform way, without bloating the aggregates interface, since the traversal algorithm is encapsulated in the iterator object. Access to the elements of the aggregate is provided by the iterator, exposing no internal structure of the aggregate. The typical division of functionality of an aggregate and an iterator can be seen in Figure 7.4.

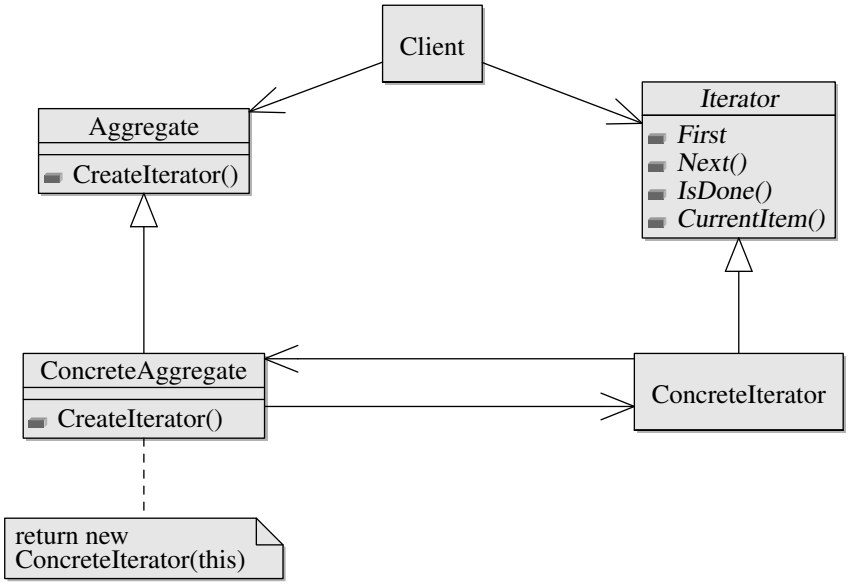

Figure 7.4: Iterator pattern UML

The functions First() ,Next(),IsDone() and CurrentItem() define the interface to any iterator, enabling a client to write code that handles any one iterator, regardless of the concrete data structure actually being traversed.

Analysis

The iterator pattern is integrated in the Scala language itself. If a collection defines the higher-order functions map, flatMap, filter and foreach it can be used with *for-comprehensions*.

The anonymous trait below shows the required operations and an example for expression. The for expression shown returns a list containing tuples consisting of a mothers name and one of her children's name.

```
trait C[A] {
  def map[B](f: A => B): C[B]
  def flatMap[B](f: A => C[B]): C[B]
  def filter(p: A => Boolean): C[A]
  def foreach(b: A => Unit): Unit
}

val persons = List(...)
// for-comprehension
for (p <- persons; if !p.isMale; c <- p.children)
yield (p.name, c.name)
---
List((Julie,Lara),(Julie,Bob))
```

The Scala compiler will translate the for-comprehension into a series of method calls of the different operations, e.g. map, filter etc., which means that the for-comprehensions are just convenient syntactic sugar. Types are automatically inferred in the for expression and the use of generics insures that no casts are needed when retrieving an element of the aggregate.

Componentization Is a non-issue in this case, since the pattern is integrated with the language.

Scala Solution

In order to implement the Iterator pattern we do not need the full power of for-comprehensions. If our collection is to be used only with a for loop, we only need to implement the foreach method.

Listing 7.3: Iterator in Scala

```
object MyCollection {
  private var items = List(1,2,3)
  def foreach(f: Int => Unit) =
   for( i <- 0 to items.length-1)
     f(items(i))
}

for( i <- MyCollection)
  println(i)
```

Again no casts or type annotations are needed when iterating the collection because of type inference and generics.

Summary The Iterator pattern is integrated in Scala and most other current mainstream OO languages, such as newer versions of Java or C#. Higher-order functions are the central feature that eliminates the need for explicitly writing iterator classes. This is a well known fact in the functional programming community [4].

Qualities of our solution:

- Deep integration with the Scala language.
- Concise: The integration allows us to use succinct syntactic sugar.

7.5 Mediator

Description Intent of the pattern:

> "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently." [9, p. 273]

Object-oriented design encourages loose coupling between classes. Message passing between these objects are essential, but if there are to many interconnections between the objects the system gets harder to maintain and reason about, because the behaviour is distributed among to many objects. In the worst case any one object ends up knowing about every other, leading to a tightly coupled system [9, p. 273]. Since objects are dependent on others in the system, they are harder to reuse in different context . As the intent states, the mediator pattern *encapsulates and centralises* the communication between the objects.

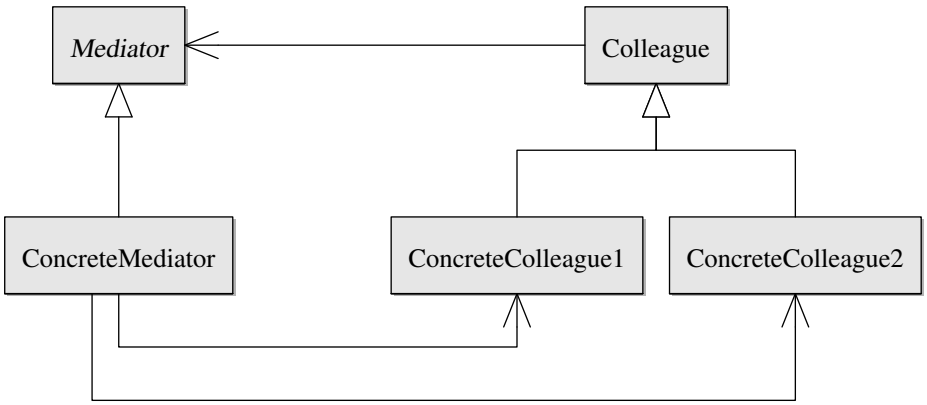

Figure 7.5: Mediator pattern UML

The Mediator class is responsible for the intercommunication between the different concrete colleagues. A colleague has a reference to the mediator which it uses for notifying the mediator when any event of interest occurs. The mediator responds by propagating the effects of the change to other colleagues, handling any requests of the involved colleagues. [9, p. 278].

Analysis

In Scala we can remove the colleagues knowledge of the mediator. This is desirable since the colleagues are freed of the pattern, which eases maintenance and improves traceability, since all pattern related code is localized. This is done by extending a colleague with a nested trait located in the mediator. This trait is then mixed in with a colleague. Using

advice on a colleagues method lets us intercept messages and react on them. A consequence of this is that the colleagues must be created through the mediator, but this is also often the case with the original solution.

Componentization If we have in mind that we want the colleagues to be unaware of the mediator, then there is nothing that stays the same between different instantiations of the pattern. The intercommunication between the colleagues change, so does their number and classes, which affects how the advice should be written. There is nothing to modularize in the mediator.

If we opted for a different solution closer to original, we could have modularized the notification mechanism as we did in the Observer pattern (7.7).

Scala solution

The classes ListBox and EntryField are our colleague classes, both of them are widgets. The DialogDirector contains a nested trait ListBoxDir that intercepts whenever our list box is clicked. When it is clicked listBoxChanged is called which will result in setting the text on our entry field with the list box's current selection. This is a simple example of object intercommunication. Notice how the colleagues are completely unaware of the mediator and therefore the pattern itself.

Listing 7.4: Mediator in Scala

```
// Widgets
class ListBox {
  def getSelection:String = "selected"
  def click = ()
}

class EntryField {
  def setText(s:String) = println(s)
}

class DialogDirector {
  // Colleagues
  val listBox:ListBox = new ListBox with ListBoxDir
  val entryField:EntryField = new EntryField

  //Directing methods
  def showDialog = ()
  // called when listbox is clicked via advice
  def listBoxChanged = entryField.setText(listBox.getSelection)

  protected trait ListBoxDir extends ListBox {
    abstract override def click = {
      super.click
      listBoxChanged
    }
  }
}

val dialog = new DialogDirector
val listBox = dialog.listBox
val entryField = dialog.entryField
```

```
listBox.click
```

Summary The limited form of AOP present in Scala enables a non-intrusive version of the Mediator pattern with enhanced locality of pattern specific code. The solution that was opted for excluded componentization.

Summarising:

- Traceability improved: Since pattern code is localised.
- Non-intrusive: Classes that participate in the pattern does not need to be modified.

7.6 Memento

Description Intent of the pattern:

> "Without violating encapsulation, capture and externalize an objects internal state so that the object can be restored to this state later." [9, p. 283]

A *memento* is an object that stores a snapshot of the internal state of another object, called the mementos *originator*. The originator is responsible for initialising the state of the memento. The state persists a snapshot of the current state of the originator and can thus be used to restore the originator to this exact checkpoint. This can be used to implement an undo mechanism etc. Only the originator should be able to store and retrieve the information from the memento [9, p. 283], but others are allowed to handle the memento, such as the Caretaker in Figure 7.6.

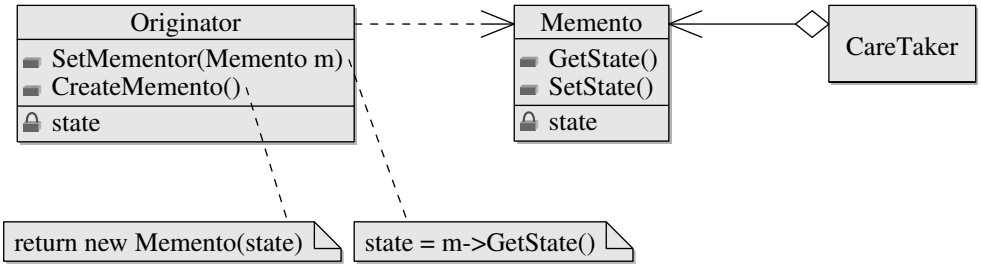

Figure 7.6: Memento pattern UML

Analysis

Most class libraries surrounding modern languages, including Java and Scala, offers some sort of serialization functionality that can be used to implement parts of the pattern [2]. "It may be difficult in some languages to ensure that only the originator can access the mementos state" [9, p. 286]. In the original C++ solution this is ensured by making Originator friend of the class Memento, thereby allowing it access to the SetState and GetState method. In Scala we can do something similar with access modifiers, the example in the solution section shows how.

23

Componentization What stays the same between different instantiations, is the role of the originator. Using an abstract type we can abstract over a concrete memento.

```
trait Originator {
  type T <: Memento

  def createMemento:T
  def setMemento(m:T)

  trait Memento {
    def getState[Originator]
    def setState[Originator]
  }
}
```

Since no boilerplate logic is present, not much is gained from extending the Originator trait.

The algorithm for copying the internal state of the originator varies between instantiations, this is somewhat analog to the issues involved in componentizing Prototype (5.4).

Otherwise the pattern is to a certain degree already available in modern languages, since most mainstream languages offers a way to serialize and persist objects. This functionality could be used in implementing the copying of the state of the originator.

Scala Solution

By nesting Memento in Originator we can use fine grained access control, e.g. getState[Originator]. Serialization can be used for saving the state, this state can then be passed to the memento and retrieved and deserialized when needed.

Listing 7.5: Memento in Scala

```
trait Originator {
  def createMemento:Memento
  def setMemento(m:Memento)

  trait Memento {
    def getState[Originator]
    def setState[Originator]
  }
}
```

Summary Class nesting and fine-grained access control enables us to restrict access to mementos state handling methods. This is done in the original C++ GOF solution with the concept of friend classes. Serialization is useful when implementing the pattern and is a part of most modern mainstream languages, including Scala.

Our solution has the same qualities as the original.

7.7 Observer

Description Intent of the Observer pattern:

> "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [9, p. 293]

The pattern is also known as publish/subscribe which indicates the structure of the pattern: An object plays the role of *publisher*, any number of objects can *subscribe* to the publisher, thereby getting notified whenever a specific event occurs in the publisher. Typically the publisher passes an instance of itself in the notification. This enables the subscriber to query the publisher for any relevant information, e.g. to be able to synchronize state.

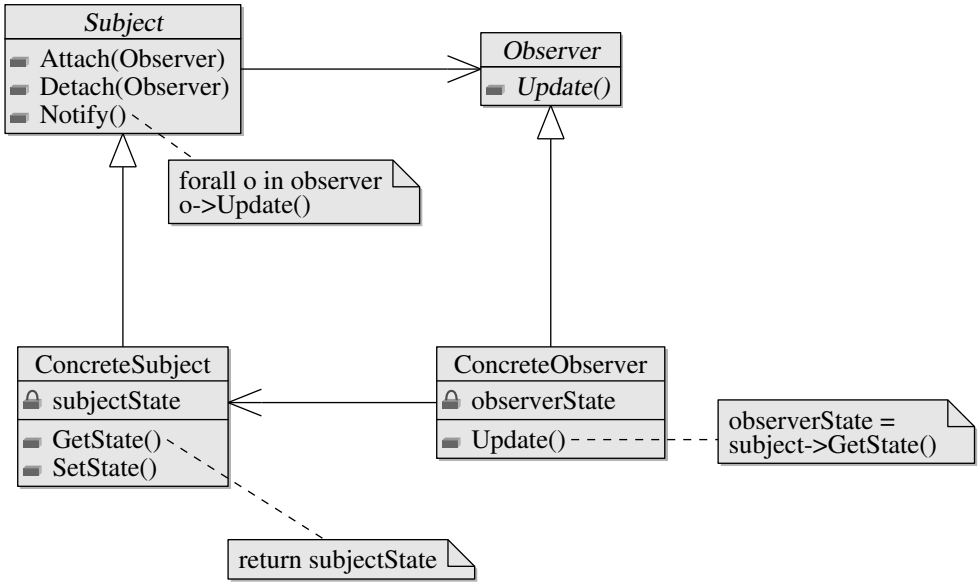

Figure 7.7: Observer pattern UML

As can be seen in Figure 7.7, the publisher, Subject, maintains a list of observers with functionality for adding and removing observers. A Notify method calls an Update function on all observers, often passing a reference to self, such that observers can query the subject for the updated state.

Analysis

A lot of boilerplate code exists in the pattern. For each instantiation we must rewrite the logic that handles the observers, this means that there is room for modularization. Using advice we can intercept any method call in the subject that should trigger a notification of the observers, this implies that we can localize all pattern related code in a trait if we are so inclined.

Componentization What differs from each instantiation is the type of observers and subject, so we need to abstract over this. Secondly we need to modularize the observer handling logic. We can define a reusable generic subject trait that captures these requirements. The trait is meant to be mixed in with any class that should function as a subject. Instead of defining an observer interface with an update method accepting a subject, we view our observers as *functions* that accept a subject.

Listing 7.6: Reusable subject

```
trait Subject[T] {
  self: T =>
  private var observers: List[T => Unit] = Nil
  def subscribe(obs: T => Unit) = observers = obs :: observers
  def unsubscribe(obs: T => Unit) = observers = observers - obs
  protected def publish = for (obs <- observers) obs(this)
}
```

The generic argument T in Subject[T] is the concrete type of our subject. In order to type the publish method, where we iterate through our observer functions and invoke them with a reference to **this**, we need an explicit self type. The type of **this** should be our concrete subject, and this will always be the case since we are composing the reusable trait with our concrete subject.

Scala solution

The following will show how we can use our componentized pattern. The class Sensor is our concrete subject. The trait SensorSubject contains pattern specific code, it extends Sensor and mixes in the reusable Subject[Sensor]. Using advice we call publish each time changeValue is called on the sensor. Placing the pattern code in SensorSubject localizes the pattern code, and enables a non-intrusive implementation. An alternative would be to integrate it in the Sensor class.

Listing 7.7: Example usage of Subject trait

```
class Sensor(val label:String) {
  var value:Double = _
  def changeValue(v: Double) = {
    value = v
  }
}

// Pattern specific code
trait SensorSubject extends Sensor with Subject[Sensor]{
  override def changeValue(v: Double) = {
    super.changeValue(v)
    publish
  }
}

class Display(label:String) {
  def notify(s: Sensor) =
    println(label + "␣" + s.label + "␣" + s.value )
}

def main(args: Array[String]) = {
  val s1 = new Sensor("s1") with SensorSubject
  val d1 = new Display("d1")
  val d2 = new Display("d2")
  s1.subscribe(d1.notify)
  s1.subscribe(d2.notify)
  s1.changeValue(10)
}
```

```
___
d2 s1 10.0
d1 s1 10.0
```

The observer class Display has a notify method that accepts a Sensor. Just as in the case of Sensor, we can make Display unaware of its role in the pattern, by creating a seperate trait with pattern specific code, that is mixed in with Display at creation time.

Summary Generics combined with self types and mixin composition enables us to write a non-intrusive componentized pattern. Using first-class functions we can get rid of the observer trait. This is perhaps a matter of taste whether we want the extra safety of a nominal interface or the less verbose, but more reusable solution.

Qualities that are present in our solution, but not in the original GOF:

- Reusability: The pattern is componentized.
- Improved traceability: It is apparent when we use the component. Pattern specific code can be localised.
- Flexibility: Observers does not need to implement a certain nominal interface.
- Implementation overhead is lowered: Observer handling logic is modularized, no nominal interface is needed.

7.8 State

Description Intent of the State pattern:

> "Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class." [9, p. 305]

The state pattern is relevant when the behaviour of an object depends on its internal state. This could be handled by conditional statements such as **switch**, but this often lead to implicit monolithic logic that is hard to maintain and extend. The State pattern offers a better way to structure state-specific code [9, p. 307]. A refactoring of such monolithic code will typically result in a concrete state class for each branch of the multi conditional logic. The Context class shown in Figure 7.8 maintains an instance of a ConcreteState that defines the current state. When clients sends messages to the context the messages are forwarded to the current state if needed.

State classes themselves can be responsible for changing to the next relevant state, this implies that the context class interface must provide operations for doing this. A reference to the context itself is often passed along when changing states, this is needed if the state classes are responsible for changing state and the context might contain relevant data for the state classes to function.

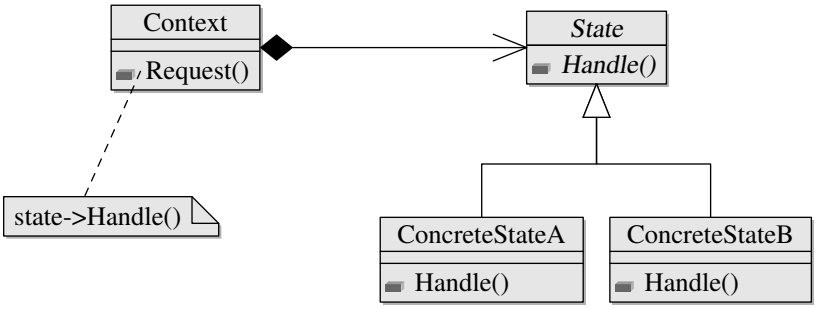

Figure 7.8: State pattern UML

Analysis

The original GOF solution uses compositon and delegation, and suffers from the self problem. As can be seen in Figure 7.8. This enables the context class to switch states dynamically, which is integral to the pattern. Since mixin composition is not a dynamic construct in Scala, it is not applicable in this scenario, we can not use it to get rid of the self problem. An alternative solution using class nesting, albeit with the same problems as the original solution, can be written in Scala. The problems are:

- State classes are coupled with the context class, since a reference to the context class is often passed at creation time.
- Boilerplate code that grows linearly for each state operation we must forward to from the context.

The usual self problem problems are present as well. But, we might argue that they are bit contrived in the sense that it is hard to imagine writing and designing the state classes without a specific context class in mind. This also lessens the severity of the first problem mentioned above.

State objects are often implemented as singleton objects [9, p. 313] which are directly expressible in Scala. The alternative solution, to be presented, uses nesting and singleton objects. This results in a more concise and localized pattern implementation.

Componentization The things that differs from each instantiation of the pattern is the interface of the abstract state class, e.g. the set of operations needed on the state classes. We cannot abstract over a set of operations in a meaningful way.

What stays the same is a reference to the current state, but this is only one line of code in our Scala solution to be presented next. Sadly we cannot modularize the boilerplate code that is needed for the message forwarding, since this is dependent on the set of operations that we cannot abstract over.

The conclusion is that we lack language expressivity to capture what should be abstracted away, at the same time there is very little boilerplate code that we can actually modularize. This leads to the conclusion that componentization is not possible.

Scala Solution

Our solution utilizes class nesting and singleton objects. The currentState variable is made private so that only the enclosing and nested classes can access it. The singleton state objects are marked private so that they are not accessible to clients.

Listing 7.8: State pattern in Scala

```
class Context {
  private var currentState:State = State1
  def operation = currentState.operation

  trait State {
    def operation
  }

  private object State1 extends State {
    def operation = { println("State1"); currentState = State2 }
  }
  private object State2 extends State{
    def operation = { println("State2"); currentState = State1 }
  }
}

def main(args:Array[String]) = {
  val c = new Context
  c.operation
  c.operation
  c.operation
  ()
}
___
State1
State2
State1
```

What we have gained with the current solution, compared to GOF, is localization of all pattern related code. The solution is more concise because of nesting and singleton objects, since the members of the context class is in scope in the state objects, and we can refer to the state objects directly, no instantiation code is needed.

There are some issues regarding extensibility. As mentioned, the forwarding code, e.g. `def operation = currentState.operation` will grow linearly. If we reach a certain number of state objects it might not be feasible to have them in one text file. The state objects can always be placed in a different file as nested objects in an enclosing trait with an explicit context self type annotation. If we want to place the state objects in *seperate* files, the self types annotations needed gets rather tiresome to write and error prone. In this case the original GOF solution is preferred.

The boilerplate forwarding code can be completely avoided in Self. Only a parent pointer to the current state is needed in the context class. The state objects would still need to know each other and the context, since they are often responsible for changing states themselves. This fact minimises the impact of the self problem, although it is not present in Self.

An alternative solution closer to the original GOF solution, that improves upon the reusability of the state classes themselves, can be written using structural types.

Listing 7.9: State pattern using structural types

```
abstract class State(s: {def changeState(s:State) } ) {
  def operation
```

```
}

class State1(s: {def changeState(s:State) } ) extends State(s) {
        def operation = println("State1"); s.changeState(new State2(s))
}

class State2(s: {def changeState(s:State) } ) extends State(s) {
        def operation = println("State2"); s.changeState(new State2(s))
}
```

Instead of expecting a specific context class, the abstract class State expects to be passed an object at creation time that satisfies certain structural requirements. In this case, only a changeState method is required.

Summary Singleton classes are often used in conjunction with the pattern, Scala provides a trivial implementation using singleton objects. The solution presented had the following qualities compared to the original GOF solution:

- Improved traceability: Pattern code is localised.
- Conciseness: As a result of nesting the state classes and implementing them as singleton objects.

An alternative solution using structural types for lower coupling between context and state classes was also presented.

An even more concise solution, that completely avoids boilerplate code, exists in Self.

7.9 Strategy

Description As described in [9], the intent of the strategy patterns is:

> "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." [9, p. 315]

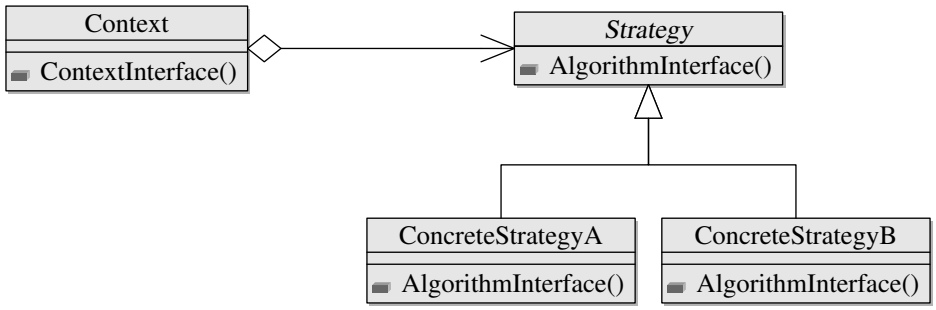

Figure 7.9: Strategy pattern UML

Figure 7.9 illustrates the concept of the strategy pattern. An abstract class Strategy defines the interface of the algorithm. The Context class maintains a reference to the current Strategy object and forwards requests from clients to the concrete algorithm. The Context

often encapsulates data structures that should be hidden from the client. The strategies might require some specific data for the computation, which is either supplied by the Context object, as an argument, or the Context object might pass a reference to itself as an argument.

Clients are commonly *responsible for creating the appropriate strategy* and passing it to the context.

Analysis

The availability of first-class functions makes promises of capturing parts of the concepts involved in the pattern easily. Combined with the unification of functions and classes and the ability to lift any method from an existing object, Scala provides much flexibility in implementing the pattern.

The original GOF solution is needed if the algorithm interface needs more than one method, that is to be called from outside the strategy, i.e. if nested methods are not enough.

Componentization If we want to provide library support for the strategy pattern, we must look at what we need to abstract over. The interface of the strategy algorithm will be different from implementation to implementation. This we can abstract away, but as we will see in the solution section, there is nothing to modularize.

We could imagine another solution where the context has a reference to the current strategy. This is according to the less tokens rule, most likely not worth modularizing.

Scala solution

The FileMatcher object from Listing 7.10 represents our context. We have a data structure private to the context, a list of files in the current directory, named filesHere. The private method filesMatching takes a strategy, matcher: String => Boolean, and returns a list of matching files.

A client selects a strategy by calling one of the appropriate methods, filesContaining, filesRegex or filesEnding. There are different ways to define the strategies themselves, depending on our needs. We can use a function literal, as done in filesContaining, which is very concise. We can use an existing method from the same class, as done in filesRegex. Or the more verbose option, defining a class, and lifting a method from the class. This also allows us to define a family of algorithms as is part of the intent of the pattern.

Listing 7.10: Strategy pattern in Scala

```
object FileMatcher{
  private val filesHere = (new java.io.File(".")).listFiles

  // matcher is a strategy
  private def filesMatching(matcher: String => Boolean) =
    for (file <- filesHere; if matcher(file.getName))
     yield file

  // Strategy selection
  def filesContaining(query: String) =
    filesMatching { x => x.contains(query) } // inline strategy
```

```
    def filesRegex(query: String) =
      filesMatching(matchRegex(query))   // using a method

    def filesEnding(query: String)=
      filesMatching(new FilesEnding(query).matchEnding) // lifting a method

    // Strategies
    private def matchRegex(query:String) =
      { s:String => s.matches(query) }

    private class FilesEnding(query:String) {
      def matchEnding(s:String) = s.endsWith(query)
    }
  }

  def main(args: Array[String]) {
    val query = args(0);
    var matchingFiles = FileMatcher.filesEnding(query)
    matchingFiles = FileMatcher.filesContaining(query)
    matchingFiles = FileMatcher.filesRegex(query)
    matchingFiles = FileMatcher.filesEnding(query)
  }
```

By only letting clients select a strategy through a method call, we can change strategy implementations without affecting the client. At first there might seem to be an extensibility problem, since we would need a method for each strategy. But in the case of inlining a strategy or using an existing method it is still less verbose than creating an entire strategy class for just a single methods sake. Alternatively we can make filesMatching public, allowing the client to supply a strategy itself.

Summary Scala gives us much flexibility in implementing the pattern. Compared to the original GOF solution our solution is:

- Succinct: We can inline strategies or use an existing method.
- Flexible: A concrete strategy ranges from function literals to classes.
- Localised: The solution shown localises all pattern code.

No component was provided.

7.10 Template Method

Description The intent of the pattern:

> "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure ."[9, p. 325]

AbstractClass in Figure 7.10 defines the skeleton of the algorithm, steps in the algorithm are refined in ConcreteClass. The method TemplateMethod() is responsible for calling the sub steps in the correct order.

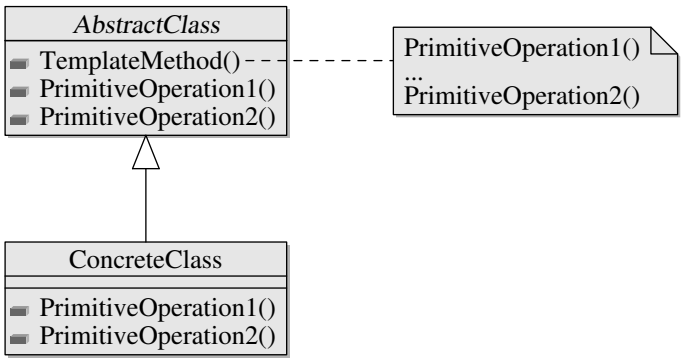

Figure 7.10: Template pattern UML

Analysis

Template Method has some essential commonalities with the Builder pattern (5.2) in that they both define an algorithm in certain steps. Template method differs in the order of the steps, which are predefined, whereas in Builder the steps taken depend on the input provided by the director. The central abstraction encapsulating the steps is in both cases a *method* belonging to some class. The refinement of the steps are realised with subclassing. The concepts involved are clearly directly expressibly in mainstream OO languages. Scala does not offer any constructs that capture these concepts better.

The unification of functions and classes does offer an alternative solution though, defining a Template Method as a class that extends some Function trait allows us to use it as a first-class function. This means that the apply method must serve as the main entry to point to the algorithm.

Componentization The number of sub steps vary from one instantiation to the next which is hard to abstract over in a meaningful way. An attempt would be to use a variable number of first-class functions, but they would have to share signatures to some degree, since we cannot define a type parameterized class that expects a variable number of types. The pattern is thus not componentizable. The concepts involved are easily expressed in the language and no boilerplate code is present, which otherwise leaves little room for componentization according to the less tokens expended rule.

Scala Solution

The solutions shows how we can define a Template Method as an extension of a Function trait.

Listing 7.11: Template Method pattern in Scala

```
trait Template extends (() => Int) {
  def subStepA
  def subStepB:Int

  def apply:Int = {
    subStepA
    subStepB
  }
}
```

The abstract methods subStepA and subStepB must be implemented in subclasses.

Summary The concepts involved in the Template Method pattern are central to object-oriented programming, such as variation in behaviour realized by subclassing. The unification of classes and functions enables us to handle an instantiation of the pattern as a first-class function. This opens up for some new possibilities, such as using a template method with higher-order functions.

Summarising the qualities of our solution compared to the original GOF:

- An instantiation of the pattern can be passed around as a function, which in the context of Scala, opens up for some new possibilities.

7.11 Visitor

Description Visitor is a well known pattern, the intent is:

> "Represent an operation to be performed on the elements of an object structure. Visitor lets you *define a new operation without changing the classes of the elements on which it operates*." [9, p. 331]

Visitor allows one to add methods that operate on an object structure without changing the classes involved. This can be very convenient if there are a lot of distinct and unrelated operations that need to be performed on the structure, in which case the structure will remain clean and non-polluted by the method implementations required for implementing the operations [9, p. 333]. Instead, the different operations can be collected in a specific visitor class, and extending the structure with new operations is straightforward.

A concrete visitor represents an operation that is to be performed on the structure. As can be seen in Figure 7.11, elements of our object structure must be equipped with an Accept(Visitor v) method. For each type of element in the structure the concrete visitor has a corresponding visit method such as VisitConcreteElementA(ConcreteElementA) that will be called in the elements accept method. This means that the method called will depend on the type of the concrete visitor *and* the type of element, as such we have a form of double-dispatch.

Typically the operation that is represented by a concrete visitor involves traversing the entire element structure, the visitor class itself can be used to accumulate state that is needed for the operation. The traversal responsibility can be placed in the structure or in the visitor classes themselves.

The element structure is often a Composite (6.3) or a collection such as a list or a set [9, p. 183].

Analysis

In the analysis of Builder (5.2) we showed how to simulate double-dispatch with pattern matching. Scalas case classes and pattern matching enables a simple implementation of the pattern that captures its intent.

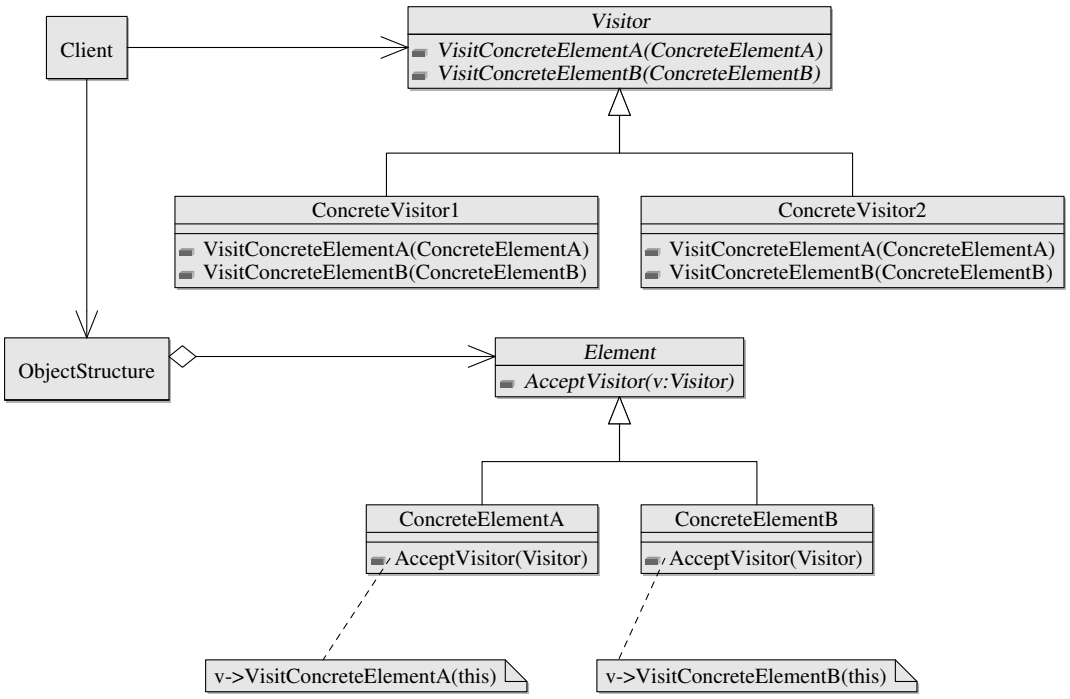

Figure 7.11: Visitor pattern UML

A concrete visitor is represented with a method that traverses the element structure using pattern matching. This solution has similar consequences as the original: The elements in the structure must be prepared to be part of the pattern, in the original solution an accept method must be defined, in the Scala solution the **case** keyword must be prefixed to the class (unless extractors (3.3.2) are used). Adding new operations are easy in both cases, define a new concrete visitor or write a new visitor method in the Scala case. Whereas adding new elements is harder, since we must change either all visitor classes or all visitor methods. In both cases the visitor can be used to accumulate state and in both cases we have to be careful with encapsulation, of the elements internals we should only expose the minimum required for the visitor to function.

Extractors can be used if we don't have source code access to the classes that we wish to pattern match (visit), or if we are worried about representation independence.

A visitor class might have several helper methods, data structures etc. Since everything is nestable in Scala, our visitor method can contain these if needed. We are off course free to define a visitor class or object if we want, that contains our visitor method(s).

Componentization Visitor has already been componentized in Scala [22].

Our presented solution is not componentizable, since there is essentially nothing to modularise between different instantiations. We cannot write reusable visitor methods, since different instantiations requires a different number of case expressions, which depends on the exact case class hierarchy.

Scala Solution

Listing 7.12 shows a simple example of an instantiation of the pattern. The Expr class hierarchy represents our composite structure. Two visitor methods are shown, prettyPrint and eval. Both are recursive functions, none of them explicitly carries state.

Listing 7.12: Visitor in Scala

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(l: Expr, r: Expr) extends Expr

def prettyPrint(e:Expr):Unit = e match {
  case Num(n) => print(n)
  case Sum(l,r) => prettyPrint(l); print("␣+␣"); prettyPrint(r)
}

def eval(e:Expr):Int = e match {
  case Num(n) => n
  case Sum(l,r) => eval(l) + eval(r)
}

def main(args:Array[String]) = {
  val e1 = Sum(Sum(Num(1),Num(2)),Num(3))
  prettyPrint(e1)
  print("\n" + eval(e1))
  ()
}
---
1 + 2 + 3
6
```

In the presented solution the traversal code is placed in the visitor method itself. An argument for putting it in the traversed structure instead, pertaining to the original GOF solution, is that we might end up duplicating traversal code. Since we are using recursive functions this is hard to avoid if we indeed have two visitor methods that traverses the structures identically. But at the same time it gives us freedom to implement complex traversals that depend on intermediate results of the object structure.

Summary Languages with double-dispatch offers an alternative way of implementing the Visitor pattern. The original Visitor pattern is inherently tied with the Composite pattern. Scalas case classes and pattern matching allows a different visitor implementation. The features involved are both a result of the unification of functional and OO programming.

Compared to the original solution:

- Conciseness: No Visitor classes are needed. Accept methods are not needed.
- Non-intrusive: In the case of using extractors.

7.12 Summary

This chapter concludes our individual analysis of the behavioural patterns catalogued in [9]. This fulfills our third goal as stated in the purpose 1. The next part will analyse the analysis as a whole, in order to answer the questions posed in the motivation.

Part III

Conclusion

Chapter 8

Related Work

This chapter will discuss related work. The work can be divided into three categories: Work that has been performed in relation to componentization. Attempts at classification of design patterns, especially in relation to language features. And finally, patterns have been explored in the context of different languages, existing or languages that have been extended to support patterns.

8.1 Componentization

Arnout and Meyer coined the phrase *pattern componentization*, and have studied the opportunities for it in the Eiffel [1] programming language [3]. They report that they were able to write reusable components for 48% of the GOF patterns. The criteria used to judge whether a componentization was a succes, were divided into the following categories: Completeness, Faithfulness, Usefulness, Type-Safety, Performance and Extended applicability. Abstract Factory was one such fully componentizable pattern, that was judged as being a success. The presented implementation of the pattern [2], is a single class, that is parameterized with a product class and a first-class function that is used for creating an instance of the product class. As the authors state:

> "An apparent limitation is that it is possible to create only one kind of product; but this is simply a matter of convention: if you need to create two kinds of product, you'll just use two factories." [2, p. 75]

With this limitation it is hard to see why this should be judged as a success, since a central aspect of the patterns intent is the creation of several products, that might dependent on each other. They have essentially written a complicated version of the Factory Method pattern, that would never survive the less tokens expended rule.

For the patterns that was described as being non-componentizable they have writen a Pattern Wizard application, that can produce a skeleton of the pattern architecture. A user of the tool is then required to fill in specific elements. Problems with this approach is discussed in Section 10.1.

Design patterns have been explored in the context of AspectJ[2] with modularity improvements in 17 of 23 cases [11]. Since AspectJ specifically targets crosscutting concerns

[1] `http://en.wikipedia.org/wiki/Eiffel_(programming_language)`
[2] `http://www.eclipse.org/aspectj/`

it is not surprising that the patterns showing the greatest improvements are those that exhibit crosscutting structures. The structures of crosscutting patterns often involves "one object playing multiple roles, many objects playing one role, or an object playing roles in multiple pattern instances" [11, p. 161]. The categories of improvements include better "locality, reusability, composability and (un)pluggability" [9, p. 167]. These improvements came primarily from the technique of inverting the dependencies between participant classes and pattern specific code, such that a pattern depends on participant classes and not the other way around. This is similar to what we have done in af few cases, such as Mediator (7.5). The pattern code is implemented with an aspect, in about half of the cases GOF this aspect is reusable.

Placing the pattern code in an aspect textually localises the code and effectively leaves the participant classes unaware of the patterns they are part of, which greatly enhances modularity. Unplugging and plugging of the pattern becomes possible since the particpants code is not mixed with pattern code and transparent composition of patterns is possible for the same reason.

As mentioned, about half of the GOF design patterns are implemented as reusable abstract aspects that must be specialized in a concrete instance of the pattern.

8.2 Classification

Design patterns have been classified in terms of how far they are from becoming actual language features [10]. *Cliches* are those patterns that essentially are macros of existing features whereas *Idiom* patterns offer a way of emulating a missing language feature. Lastly, we have *Cadets* which are abstractions "that are not yet sufficiently mature or important enough to be language features" [10, p. 118]. These are the real patterns, in that they capture essential know-how, not directly present in any language. Design patterns are seen as a mean for expressing higher-level abstractions not expressible in the language itself.

The authors argue that the lack of taxonomies for Design patterns is the result of bundling the patterns with "intent", since intent "transgresses the boundaries of exact science, which is limited to "how" rather than "why" questions"[10, p. 120]. The language oriented view that they propose ignores intent, which may help understand and illuminate the interplay between patterns and languages, but have several other problems instead. The proposed categories does not seem stable since they will depend on ones view of a language, e.g. when is a feature missing? Perhaps the language in question belongs to a different paradigm than the language containing the feature, as such they have different ways of doing things. What is the line between Cliches and Cadets? An experienced programmer will probably regard some patterns as Cliches, whereas a novice will regard them as Cadets.

The success of design patterns might turn out to be a two-edged sword, the proliferation of identified and cataloged patterns can eliminate some of the benefits originally gained [1]. The sheer number of design patterns can make it too hard to find and use the encapsulated experience and will make the common vocabulary too large to be easily comprehended. In their analysis of the original GOF patterns, Agerbo and Cornils define several criteria and categories in order to filter the design patterns that should be cataloged. Amongst the categories are Fundamental Design Datterns FDP, that are independent of any implementation language and are not covered by any language con-

struct. Language Dependent Design Patterns LLDP, should be partioned into what language construct they essentially make up for. FDPs are the essential design patterns that should be part of the common vocabulary, others should be dismissed in order to keep the catalog minimal.

12 of the GOF patterns are identified as FDPs whereas most of the other are LDDPs. The implementation language is Beta [14] and the missing language constructs the LDDPs make up for, according to the authors, include nested classes in the case of Facade and Singular objects in the case of Singleton. The article goes on to discussing how to deal with the problems of traceability and reusability, their solution is to try and componentize the fundamental patterns, which is done for 10 of the 12 FDPs, the primary constructs involved are nested classed and virtual classes. The usefulness of the componentized patterns are not discussed.

Comparing [1] and [10], we can identify FPDs with Cadets and LDDPS with Idioms.

8.3 Languages and Patterns

Patterns have been explored in the context of *dynamic languages*. More specifically Scheme with a library of functions and macros to provide OO facilities [24]. It was discovered that some *patterns disappear since some language constructs captured them.* Others were greatly simplified or almost disappeared mainly due to first-class functions. Multiple dispatch was used in writing a more elegant Visitor solution. Some patterns were identified as universal programming concepts, such as Template Method. The work is an example of how much pattern implementations depends on the used language and as such lies much in line with this thesis.

Bosch creates [5] a layered object model called LayOM with support for design patterns through such constructs as “layer” and “state”, which permit to represent pattern concepts more directly. LayOM can be extended with new components and layer types. LayOM classes are translated to C++ code, it is essentially a code generator approach.

8.4 Summary

Earlier attempts of componentization has not been an overall success. Especially in the case of Eiffel [2], the solutions are often not satisfactory.

Attempts at pattern classification has been discouraging. All suffered from highly subjective categories.

The expressitivy of dynamic languages, enables concise implementation of patterns. But this comes at the cost of no static guarentees. Aspect-oriented programming is beneficial for implementing design patterns, as it lets us capture the cross-cutting concerns that are often present in patterns. Further, we can invert the dependence between pattern and participiating classes. Using advice 3.6 in Scala, we have done this in a few cases, resulting in non-intrusive implementations.

Chapter 9

Results of Analysis

Based on our analysis from Part II of the individual patterns , we will now proceed with a presentation of the findings in the analysis as a whole. These findings will be the base upon which we answer the questions stated in our motivation.

We will start with an overview of the Scala solutions, indentify which features were essential and whether the unification played a role. Further, we will relate the features used to the purpose of the pattern and discuss the number of levels of indirection present in the new solutions.

The results of the componentization are presented, answering such questions as: Where the constructs identified as being essential when writing components instrumental in creating our componentized patterns? Which general problems did we face in the componentization process?

Finally, we will present the findings regarding the implementation problems identified with design patterns and the new solutions. The chapter ends with a discussion of the second principle of GOF, "Favor object composition over class inheritance", in the context of our Scala solutions.

9.1 New Solutions

Table 9.1 shows for each pattern, the language features that was essential in enabling the new solution. The pattern column is divided into the three purpose categories: Creational, Structural and Behavioral. A • indicates that the feature was essential whereas a ◦ shows that the feature was used, but did not play a central part. E.g. a Facade is often implemented as a Singleton but it is not essential in the implementation of the pattern.

Note that **Advice** (3.6) is not a language feature as such, but rather an idiom, we feel that it makes sense to keep it distinct from the **Mixin Composition** category, since it is a valuable technique. **Access Modifiers** refers to the usage of the keywords **private** and **protected**, especially in the contex of nested classes and more fine-grained access control such as **private**[Foo] etc.

Purpose and features When reading Table 9.1 it is quite apparent that there is a connection between the features used and the patterns purpose (2.2). First-class functions are almost exclusively used in *behavioral* patterns, whereas abstract types and self types are mainly present in creational and structural patterns. This is quite natural since first-

	Abstract Types	Advice	Access Modifiers	Case Classes	Pattern matching	Class Nesting	First-Class Fun	Generics	Lazy	Mixin Composition	Self Types	Singleton Obj	Implicits
Abstract Factory	•		•			•				○	○	○	
Builder													
Factory Method	•									•			
Prototype													○
Singleton												•	
Adapter										•	•		
Bridge										•	•		
Composite				•	•								
Decorator													
Facade	•		•			•						○	
Flyweight							○	•				○	
Proxy									•				
Chain Of Responsibility								•		•			
Command							•						
Interpreter				•	•								
Iterator							•	•					
Mediator		•	○			•				○			
Memento			•			•							
Observer		○					•	•		•	•		
State						•						•	
Strategy							•						
Template Method							○						
Visitor				•	•								

Table 9.1: Central features

class functions abstract over behaviour while abstract types and self types are useful when expressing structural relationships between classes.

The most used features are: Mixin composition, first-class functions, class nesting and generics. This is not surprising, since they are all very general constructs.

Levels of indirection in new solutions Four implementations of behavioral patterns using first-class functions have *fewer levels of indirection* than their GOF counterpart, as can be seen in Table 9.2. In Command callbacks are called directly. In Iterator there is no need for an explicit iterator object. In Observer there is no need for an observer trait with a`notify` (7.7) method, the function is called directly, and in Strategy we also call the current algorithm directly. This means that first-class functions regarded as a construct is close to the main concept involved in these patterns.

Regarding levels of indirection in general, we see that in 10 out of 23 patterns, the Scala solutions exhibit *fewer levels*, or other simplifications, than their GOF counterparts. Besides first-class functions, it is mainly due to mixin composition, which helps in eliminating the self problem. Disregarding the solution as a whole, a pattern void of the self problem, will have a level less than a pattern that suffers from it.

In the case of first-class functions the levels of indirection still exist deep in the Scala language itself, because of the way they are implemented, i.e. as classes with an `apply` method. But this is hidden from the user, with the use of syntactic sugar.

Fewer levels, suggests a more expressive language regarding the concepts involved in the GOF patterns. This in turn suggests that *implementation overhead is lowered*.

Unification First-class functions are clearly beneficial when implementing design patterns. An important aspect is that they, in order to have high-value in an OO setting, must be integrated with the class construct. Taking an existing method defined in a class and passing it, is used in several patterns, which shows the value of this integration.

Case classes and pattern matching played a major role in three patterns: Composite, Visitor and Interpreter. This object-oriented version of algebraic data types and pattern matching, allows a fairly straight-forward version of the patterns to be implemented, while maintaining the solutions ability to participate in other patterns. The utility of case classes and pattern mathing may in some cases be obstructed by the problem of non-exhaustive matches (3.3.2). This depends on the exact scenario, such as whether it is feasable to use **sealed** classes or providing a meaningful default case. This must be judged individually in each case.

The third unification, that of modules and objects, has played a lesser role when compared to the other unifications. Path-dependent types are only fully used in Abstract Factory.

9.2 Result of componentization

Table 9.3 shows the result of the componentization effort. **Component** indicates whether the pattern compentization succeded. • marks a fully componentized pattern, while ◦ marks a partly componentized pattern. **Less tokens expended** indicates whether the rule was deemed in effect. Finally, **Problem** indicates whether, in case of an unsuccesful componentization, we had an abstraction problem or a modularization issue.

Pattern	FLoI	Other Simpl.	Reason
Abstract Factory			
Builder			
Factory Method			
Prototype			
Singleton		•	Built-in
Adapter	•		No message forwarding to adaptee
Bridge	•		No message forwarding to implementors
Composite			
Decorator			
Facade			
Flyweight			
Proxy		•	The use of lazy keyword
Chain			
Command	•		Callback invoked directly
Interpreter			
Iterator	•		No iterator object
Mediator	•		No event mechanism, advice is used instead
Memento			
Observer		•	No observer trait
State			
Strategy	•		No message forwarding to strategy object
Template Method			
Visitor		•	No accept method or visitor class hiearchy

Table 9.2: Fewer levels of indirection and other simplifications

Only two patterns were properly componentized: Observer and Chain of Responsibility. Flyweight was partly componentized. In 6 cases if was deemed fruitless according to the **less tokens expended** rule.

In 11 cases something fundamental in the pattern itself eluded the componentization process. Abstracting over the *set of members in a class* in a meaningful way, is not possible in Scala. The Template Method pattern is a good example of this. The number of steps in the pattern depends on the context, making it impossible to provide a general component. An unsatisfying solution would be to provide n classes with n substeps. This technique is used by Scala itself. There exists 23 function traits, each of them represent a function with n arguments, where $n \leq 23$. These are needed because of the unification of functions and classes, since we cannot abstract over the number of arguments in the class construct itself.

Communication between objects is the second componentization showstopper. Mediator and Facade are both examples where some abstraction over the communication of objects is needed in order to realize the componentization. Seeing that we cannot abstract over the set of members in a class, abstracting over the communication of objects is clearly not possible either, since the communication is characterized by, among other

things, the set of communicating methods.

Finally, two of the patterns were built-in: Singleton and Iterator. Part of Memento is built-in, since the Java standard library provides serialization functionality that can be used to persist an object.

The features in Scala that was identified as being essential in writing components (3.7), that is, abstract type members, explicit self types, and mixin composition, was used in varying degrees in the implementations. Regarding the two componentized patterns, Observer and Chain of Responsibility, mixin composition was essential and used in both. Self types was vital in Observer but not used in Chain of Responsibility. Abstract type members were used in neither. In the case of Chain of Responsibility, we could have replaced the use of generics, with an abstract type member. This was not possible in Observer, since the type parameter would not be in scope for the explicit self type. From a general GOF patterns componentization point of view, these features were arguably not essential, since they did not allow us to componentize other patterns than those mentioned. But we could speculate that as soon as we start to refine the pattern implementations themselves, e.g. an Abstract Factory implementation where abstract types play a rule, they will play a bigger role.

	Component	**Less tokens expended**	**Problem**
Abstract Factory		•	set of products
Builder			set of methods
Factory Method		•	
Prototype			shallow and deep copy
Singleton	built-in		
Adapter			communication between objects
Bridge			communication between objects
Composite		•	
Decorator		•	
Facade			nothing to modularize
Flyweight	∘		
Proxy		•	
Chain	•		
Command		•	
Interpreter			nothing to modularize
Iterator	built-in		
Mediator			communication between objects
Memento	(serializable)	•	
Observer	•		
State			sef of operations
Strategy		•	
Template Method			set of substeps
Visitor			nothing to modularize

Table 9.3: Result of componentization

9.3 Design Pattern Problems

Traceability Two possible solutions to traceability, was discussed in Section 2.6. Componentization and strong locality. The weaker notion of a close connection between concepts in the pattern and language constructs as an indicator of improved traceablity, and whether this has an impact on traceability was also mentioned.

Traceability has been *improved in the cases where componentization has succeded*. The usage of the two componentized patterns: Chain of Responsibility and Observer, and the partly componentized pattern Flyweight, is quite apparent in the source code. Since the traits involved are named by their roles in their respective patterns.

A few pattern implementations where more localized than their GOF counterpart. These include Mediator, State and Strategy (besides the componentized patterns, which also has stronger locality). Localization has the potential of improving traceability, since the pattern code is no longer scattered across several source code files. A cryptic implementation, although localized, can still have bad traceability.

Traceability might be improved when constructs in the language are closer to the concepts in the pattern. It could be argued that a programmer that is knowledgeable of the constructs and patterns, who is reading source code that contains implementation of a pattern using these constructs will have an easier time spotting the actual pattern. Take Command as an example. In the original solution, the concept of a *binding between receiver and a request* was represented by a command object that stored this binding. Some sort of naming scheme identifying this as a command object would have to be in place, or else this would be just another object in the system. Compare this to the less involved notion of *storing an explicit function or method belonging to some class directly in a variable*. Whether this improves traceability is highly subjective, and is of no use as a metric when trying to improve traceability.

Reusability Reusability was only improved in three cases where we managed to provide a componentized pattern.

Implementation overhead Seeing that some patterns are built-in, others are components and especially behavioral patterns exhibits fewer levels of indirection, *implementation overhead has in general been lowered*. But there are two patterns with higher implementation overhead, because of the usage of abstract types. These are Factory Method and Abstract Factory. We could argue that this is just an initial overhead, that will be compensated for in subclasses, since there are more opportunities for code reuse.

Self problem The use of mixin composition in the implementations of Bridge and Adapter removed the self problem. This was also the case for the rejected Decorator solution. Finally, our virtual Proxy solution avoided the problem altogether with the use of laziness. These solutions still lack the complete flexiblity of the original GOF solutions, indicating that they may not be adequate for all scenarios. Reverting back to the usage of object composition will reintroduce the self problem.

9.4 GOFs Second Principle

The second principle "Favor object composition over class inheritance" [9, p. 20] was identified as being in effect in the following patterns: Adapter (object), Bridge, and Decorator. The use of composition in a pattern does not necessarily mean that the principle is in effect. Take Composite as an example, it is hard to imagine an implementation that uses classic inheritance as a replacement for composition. Whereas in the original Decorator solution, an inferior alternative solution using inheritance is discussed. The keyword here is *reuse*.

Is the principle still valid ?

Our Adapter solution no longer uses composition, but mixin composition instead. It has all the benefits of the original object Adapter solution, except that we cannot change adaptee at runtime. This is analog to our Bridge solution, where we cannot change implementation at runtime, but we escape the proliferation of classes that the standard inheritance mechanism would impose. The ability to change runtime binding was deemed so essential in the Decorator pattern that the suggested solution was rejected.

Since both our Adapter and Bridge solutions lack the dynamic features of the orginial solutions, which are essential in certain scenarios, we conclude that the principle is *weakened but still very much alive*.

9.5 Summary

The features used in the new solutions was related to the purpose, a dimension in the design space, of patterns. This showed a strong correspondence between first-class functions and behavioral patterns. In 4 implementations fewers levels of indirection exists, since first-class functions matches the concepts in the pattern. This in turn has a positive effect on implementation overhead.

Traceability was in general not improved. Only in the cases where reusability was improved because a component was provided. The self problem was removed from three pattern implementations, although the solutions were not applicable for all scenarios. From which we can infer that the 2nd principle of GOF is still valid, although weakened.

The language constructs identified as being essential when writing components, were instrumental in the componentization of three patterns.

The unification of ADTs and class hierarchies and the existence of pattern matching, proved usefull in the implementation of three patterns.

This concludes our third goal.

Chapter 10

New Features Discussion

This chapter will present and discuss language features not present in Scala but relevant for the implementation or componentization of patterns in the context of Scala.

10.1 Static Metaprogramming

Static metaprogramming capabilities could allow us to abstract over the set of methods in a class. Several of the patterns that proved impossible to componentize in Scala, contains little or no boilerplate code, such as Template Method. This means that there is essentially nothing to modularize, only the roles in the pattern, is present in all implementations.

In the case of Template Method, assume that we had static metaprogramming capabilities in Scala, that allowed us to define a metaclass that could produce a class, at compile time, with a variable number of methods. Assume also that we had the ability to invoke a variable amount of methods. A reusable Template Method *meta trait*, parameterizable by the number of substeps N, could look like this.

```
trait TemplateMethod<N> {
  def step1 .. stepN

  def algorithm() {
    step1() .. stepN-1()
    step_N
  }
}
```

The trait TemplateMethod contains N methods, step1 .. stepN. These are called in the method algorithm, the return value of the last method is the result of the entire algorithm. The trait essentially captures the protocol of the pattern, i.e. execute step1 .. stepN−1 and return result of stepN. When using this trait we would have to specify N and implementation code for the methods. Leaving types and type annotations aside, though very important but not for the point we are trying to make; using this trait would result in generic names for the steps in the algorithm and the algorithm itself. The usage of the pattern would be explicit, but having several instantiations of the pattern using this trait in play at once could severely hurt code comprehensibility. The only logic the pattern has mod-

ularized is in a sense the order of the substeps, but these are not defined yet, so we have essentially modularized nothing.

We could imagine more complex meta traits, that involved control structures etc., in which we could benefit by modularization. But this is irrelevant for the discussion, since we are contemplating on how to modularize patterns that contain little or no boilerplate code. If we were to specify the names of the methods also, which would alleviate the problem of code comprehensibility, the abstraction would likely not survive the less tokens expended rule.

Can we generalize this analysis to other patterns that we could not componentize, but only those with little boilerplate code were only the conceptual roles stays the same between different instantiations. These include: Adapter, Bridge, Facade, Mediator and State. Using metaprogramming we could try and *capture the protocols* in these patterns. Again, little would be gained, since no boilerplate code exist: These pattern meta traits would just be *skeletons* and depending on our strategy, with our without meaningfull (in the domain specific context) names. Another issue is that of introducing metaprogramming syntax, which must be learned by the programmer, which in the case of patterns with no boilerplate code, provides few benefits.

There are a few positive sides though. Traceability would be improved and the meta traits could perhaps be instrumental in learning patterns, since the programmer is forced to fill in relevant details for the specific instantiation of the pattern.

We could also imagine a codegeneration tool, external to the language. This tool could help in generating a skeleton for a pattern, with meaningful names supplied by the programmer. Such tools already exist for some languages [3]. This approach has some issues with composition of patterns. Regarding the external tool, if a class participates in several patterns, the tool must somehow support this, or else the programmer will end up generating a skeleton for a certain pattern, and then implementing the other pattern in the hand, in the generated code, which seems unsatisfying.

The conclusion is that regarding patterns with no boilerplate code, neither approaches seems a fruitful path to explore.

10.2 New constraint

The following section will discuss extending Scala with a *new constraint*. The term is adopted from a similar feature, present in C# [1]. The **new** constraint in C# allows one to specify that a type argument, in a generic class declaration, must have a public parameterless constructor. This is done with a *where* clause: **class** ItemFactory<T> where T : **new**(). A similar where clause, although applicable in a broader sense, not just with generic type arguments, was first introduced in the programming language CLU [13] authored by Barbara Lisskov.

As discussed in Factory Method (5.3), it is not possible to instantiate an abstract type, even though it would be beneficial to be able to do so in the pattern. Let us assume that we could specify that an abstract type, or type parameter, was a subtype of a certain other type *and* was guaranteed to have a default parameterless constructor. We use <:: to express this, instead of <:.

[1] `http://msdn.microsoft.com/en-us/library/sd2w2ew5(VS.80).aspx`

```
trait Application {
  type D <:: Document
  var docs = List[D] ( )
  def newDocument = {
    val doc = new D // now legal!
    docs = doc :: docs
    doc.open
  }
}
```

This would get rid of the factory method itself, which would also be beneficial for Abstract Factory (5.1). But what about constructor arguments? These would clearly be needed in Abstract Factory, were some products might depend on others.

```
trait WindowFactory {
  type aWindow <: Window
  type aScrollbar <: Scrollbar

  val scrollbar:aScrollbar = new aScrollbar
  val window:aWindow = new aWindow(scrollbar)

  abstract class Window(s:aScrollbar)
  abstract class Scrollbar
}
```

The question is whether the presence of such an constructor could be inferred by the upper bound Window. If not, we would have to specify the number and types of arguments somehow, in the constraint itself. In this case we could just as well use a Factory Method, which we could think of as an idiom for this feature. We conclude that factory methods serves us well, so the feature is not needed.

10.3 Dynamic inheritance and True delegation

Dynamic inheritance and true delegation are related concepts. As composition mechanisms regarded, they are both void of the Self problem. Dynamic inheritance is said to exist in Self, because of the presence of true delegation. Either of their presence in Scala, would be beneficial for the implementation of Design Patterns.

Delegation Sugar Syntactic sugar has been exploited to a great degree in the Scala language. Is is possible to use syntactic sugar to ease the implementations involving delegation, such as avoiding the boilerplate code in the State pattern (7.8)?

We could imagine a *delegate* keyword, that when combined with a field, would result in the compiler automatically inserting methods that handle *trivial forwards* (2.5). The field would have to be mutable inorder to provide any real benefit, since we must be able to change delegatee at runtime. What remains are the issues of *self sends* and *identity*. These cannot be solved with syntactic sugar. Solutions to both of them require that *self* is bound to the original receiver. This requires that the rules for method lookup would have to change dramatically in the language.

In the programming language Self, the method lookup rules are such that this type of delegation is possible. Methods are represented as prototypes of activation records and are cloned upon instantiation. In this process self is bound to the original receiver, thereby enabling true delegation. As such, the enabling of this, permeates the whole language, from syntax to runtime. Changing the rules for method lookup in Scala is far from non-trivial and would require a major overhaul of the language. Without new rules for method lookup, the only problem delegation sugar solves is that of trivial forwards. We conclude that delegation sugar is not worth the trouble.

Dynamic mixins Regarding dynamic inheritance in Scala, we could restrict ourselves to only being able to change the implementation of a *trait consisting only of methods*. I.e. we restrict ourselves from adding new traits to an object. E.g. in the composition TextView **with** BorderDecorator we cannot extend with further traits, but we can change exactly which trait the decorator trait (BorderDecorator) should belong too, as long as it is a subtype of the original and only consists of methods. Implementation wise, this should be feasible, since all we essentialy need is to change the virtual table that is used for method lookup.

In two of the three cases that still benefits from the 2nd principle: Decorator and Bridge, dependending on the exact scenario, this is all we need to get the dynamic flexibility of the original GOF solution back. In neither of our examples the traits carry state. Sadly, it is not a usable solution for Adapter, since by definition the adaptee is almost always some class. Often a class we do not have access too, which makes it more likely to have state.

The more general case of being able to change class of any object, is much harder to implement, since the heap must be modified when e.g. instance variables are added or removed.

In order to switch a specific mixin in an object, we must at the point of change know whether or not the corresponding trait is present. This can be achieved with a subtype check. An alternative is to only let the trait itself, or a class change one its traits. This requires no subtype checks.

We could imagine a private method **def** changeMixin(NewTrait). An alternative implementation of State with the use of dynamic mixins:

```
val context = new Context with State1

trait State1 {

  def operation() = {
    println("state 1")
    this.changeMixin(new State2)
  }
}

trait State2 {

  def operation() = {
    println("state 2")
    this.changeMixin(new State1)
  }
}
```

Changing the *class of an object* at runtime is not a new concept. A safe version of object re-classification exist in the $Fickle_3$ [7] language. *Effect annotations* on methods are used to guide the static analysis. The effect indicates whether the method might change the class of an object, whether it be self or some object in scope. The effects are set of pairs $\{c_1 \Downarrow c_1', \ldots, c_n \Downarrow c_n'\}$, indicating that the method may perform any re-classification, at runtime, from a subclass c_i' to a subclass c_i' [7][p. 2]. The entire setting is simpler than that of Scala, since we are only dealing with single inheritance in a Java like language.

Nested Mixin-Methods exists as class attributes in Angora [23]. If a mixin attribute is present in a class, the mixin can be applied to the class. Though angora mainly has a focus on *constraining* the shape of inheritance hierarchies, thereby preventing changes in semantically dubious directions, such as a CircleRectangle class in a graphics hierachy. Mixin methods are realised with message passing and can be applied *dynamically* [23][p. 214]. This means that we can swith the implementation of an already defined mixin attribute. Mixin methods are untyped [23][p. 217], the implementation of Angora runs on top a Smalltalk implementation. This would be unacceptable in a Scala setting.

If a safe version of dynamic mixins was present in Scala, it would improve two implementations that lack the dynamic flexibility of the original solutions, to such a degree that the 2nd principle would have nothing to offer in these cases.

10.4 Summary

Static metaprogamming capabilites was discussed as a potential abstraction mechanism for capturing patterns that had otherwise proven impossible to componentize in Scala. The idea was rejected: Adding metaprogramming syntax just for the sake of componentization of patterns with no boiler plate is not worth the extra complexity in the language, since the gain is very little.

A new constraint, benefical for creational patterns, was rejected. The discussion showed that a Factory Method served well as an idiom.

Finally, delegation sugar and dynamic mixins was presented. Delegation sugar was not viable as a means to avoid the self problem when using delegation. Dynamic mixins on the other hands, were.

This chapter concludes our last goal. The following will conclude the thesis.

Chapter 11

Conclusion and Perspectives

"Brevity is the soul of wit.."
(Shakespeare's *Hamlet*, 1603)

This chapter will conclude the thesis. Initially, the current state is summarized, followed by perspectives.

11.1 Current state

This section will review the current state of the thesis. The purpose of the thesis was formulated as four successive goals, in Chapter 1. Each of the four goals have been accomplished. The goals are stated individually below, accompanied by a description of the current state relating to the goal.

1. Identify known implementation problems with Design Patterns:

   - Based on [5], the notions of traceability, reusability and implementation overhead in the context of design pattern implementations were presented and discussed (2). An analysis of the consequences of the self problem [12] in a concrete example, resulted in the identification of the problems of: Trivial forwards, self sends and identity.

2. Give an overview of the novel features in Scala:

   - Chapter 3 presented the novel constructs, including: Mixin composition, abstract types and self types.

3. Analyse a collection of Design Patterns in the context of Scala. For each pattern provide an improved implementation or a component.

   - Each GOF pattern was analysed in Part II. For each of the patterns we succeded in providing an improved implementation, at least for certain scenarios. Two patterns were succesfully componentized (7.1)(7.7), a third partly (6.6).

4. With the questions from the motivation in mind, present and identify findings in the analysis as a whole:

- Did the novel constructs in Scala enable us to write reusable patterns? To a limited degree. Mixin composition and self types were essential. Abstract type members less so (9.2).
- When should a pattern be componentized? When the pattern has a sufficient amount of boilerplate code. This is discussed in Section 4.2 and 10.1.
- Solved implementation problems: Traceability was improved in three cases, so was reusability. About half of the implementations exhibited fewer levels of indirection or other simplifications, resulting in less implementation overhead. The self problem was solved in two cases (9.3).
- Are the functional features present in Scala useful when implementing design patterns? First-class functions were very usefull regarding behavioral patterns. Case classes and pattern matching allowed straightforward implementations of Composite, Visitor and Interpreter. (9.1).
- The 2nd principle of GOF is still valid, though weakened. If dynamic mixins were present, the principle would in the context of our catalog of patterns, have to be reevaluated (9.4).
- Features not present in Scala but beneficial for the implementation or componentizatin of patterns: Static metaprogramming was discussed but rejected (10.1), so was a "new constraint" (10.2). Since delegation sugar did not solve the problem, dynamic mixins was preferable (10.3).

11.2 Perspectives

Regarding the constructs that was idenfied as being essential when writing components: Their apparent uselessness in the majority of cases, might stem from the fact that design patterns as a whole are a different beast altogher than components. A component without modularized logic is not much of a component, several of the patterns contains no boilerplate to modularize. This fits well with the fact that static metaprogramming was rejected as a useful abstraction mechanism when componentizing patterns with no boilerplade code.

The notion of a design pattern is flexible: Some serve as reminders of object-oriented techniques, such as Template Method. Others are clearly components, with internal logic, such as Observer. We believe that this flexibility is essential, since it gives us a non-constrained metalevel to discuss design, programming and programming languages.

The second principle of GOF "Favor object composition over class inheritance" is still valid in Scala. Object composition has some major shortcomings: The delegation used is implemented with message forwarding, which results in less reusable classes, since delegate and delegatee has a knowledge of each other. Further, it introduces the self problem. This suggests that a composition mechanism with the dynamic capabilities of object composition, but void of the self problem, would be very benificial to have in Scala.

What happens to the informal qualities (2.4) when a pattern is available in a library? They are probably strengthened, since the usage of the pattern is more visible. Indeed, discussions between developers centered around the pros and cons of different software libraries are not unheard of.. Since the pattern is available in a library, there is a good change more developers will get to know it through all the different channels libraries are exposed. Such as modern IDE's and their integrated help, internet sites, books etc.

Do we loose all the informal qualities when a certain design pattern suddenly gets “invisible” or perhaps more implicit in a design, since the language provides support for it at a basic level? We should in a sense, since the pattern is now deeply integrated in the language, the solution provided by the pattern is a none issue. To give an example, OO techniques could be patterns in a procedural language. The intent of the “object pattern” might be a solution to “How do we encapsulate and control the access to state?”. This pattern is so deeply integrated in OO programming that the pattern quite naturally does not show up in any design pattern textbooks.

By providing better language abstractions that can handle the most common design patterns, one could hope that a new level of patterns might emerge. Pawing the way for the writing of increasingly more capable software at a lower cost.

Bibliography

[1] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. *SIGPLAN Not.*, 33(10):134–143, October 1998.

[2] Karine Arnout and Bertrand Meyer. Pattern componentization: the factory example. *Innovations in Systems and Software Technology: a NASA Journal*, 2(2):65–79, 2006.

[3] Katrine Arnout. *From Patterns To Components*. PhD thesis, Swiss Institute of Technology, Zurich, 2004.

[4] Henry G. Baker. Iterators: signs of weakness in object-oriented languages. *SIGPLAN OOPS Mess.*, 4(3):18–25, 1993.

[5] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11:18–32, 1998.

[6] Gilad Bracha and William Cook. Mixin-based inheritance. pages 303–311. ACM Press, 1990.

[7] Ferruccio Damiani, Sophia Drossopoulou, and Paola Giannini. Refined effects for unanticipated object re-classification: Fickle3 (extended abstract). In *In ICTCS'03, LNCS 2841*, pages 97–110. Springer, 2003.

[8] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Resusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[10] Joseph Gil and David H. Lorenz. Design patterns vs. language design. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 108–111, London, UK, 1998. Springer-Verlag.

[11] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, November 2002.

[12] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[13] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in clu. *Commun. ACM*, 20(8):564–576, 1977.

[14] Ole Lehrmann Madsen. Towards a unified programming language. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 1–26, London, UK, 2000. Springer-Verlag.

[15] Martin Odersky. Scalable component abstractions. In *In Object-Oriented Programming: Systems, Languages, Applications (OOPSLA). ACM*, pages 41–57. ACM Press, 2005.

[16] Martin Odersky. The scala experiment – can we provide better language support for component systems? Invited talk abstract, POPL, 2006. `http://www.ist-palcom.org/publications/files/The%20Scala%20Experiment.pdf`.

[17] Martin Odersky. The scala experiment – can we provide better language support for component systems? Invited talk, 2006. `http://lampwww.epfl.ch/~odersky/talks/popl06.pdf`.

[18] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[19] Martin Odersky and al. The scala plugin for eclipse. Technical report, EPFL Lausanne, Switzerland, 2006.

[20] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, 2003.

[21] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 2008.

[22] Bruno C.d.S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. *SIGPLAN Not.*, 43(10):439–456, 2008.

[23] Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested mixin-methods in agora. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 197–219, London, UK, 1993. Springer-Verlag.

[24] T. Sullivan. Advanced programming language features for executable design patterns "better patterns through reflection" year = 2002.

[25] Mads Torgersen. *Unifying Abstractions*. PhD thesis, Aarhus University, Denmark, 2001.

[26] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, volume 22, pages 227–242. ACM Press, December 1987.

Chapter 12

Glossary

AOP Aspect-oriented programming.

AST Abstract syntax tree.

DSL Domain specific language.

EBNF Extended Backus-Naur Form.

Idiom A programming idiom is an expression of a simple task or algorithm that is not a built-in feature in the programming language being used. Or, conversely, the use of an unusual or notable feature that is built in to a programming language.

Internal DSL A domain specific language that is embedded in some host language, using only the features of this language. Allowing one to use the existing tool chain available for the host language.

JVM Java virtual machine.

Nominative type system Class of type systems, in which type compatibility and equivalence is determined by *explicit declarations* and/or the name of the types.

OO Object-oriented language.

SML Standard ML.

Structural type system Class of type systems, in which type compatibility and equivalence are determined by the type's *structure,* and not through explicit declarations.

Token A token is a categorized block of text, resulting from the process of lexical analysis, which is a sub-task of parsing. An expression such as `sum = 3+5` contains the tokens; `sum`, `=`, `3`, `+` and `5`.