Facilitating Software Reuse Through Blobs and Blobjects

Niklas Björkén
Maria Patrickson

Department of Computer and Systems Sciences
Stockholm University / Royal Institute of Technology

June 10, 2008

This masters thesis corresponds to 30 credits for each author.

Abstract Software reusability is closely related to software quality, cost of development and time spent on software projects. Despite this fact, reusability is considered difficult to achieve and has to be explicitly designed for. Programming paradigms and architectures of today are not enough to accommodate the need for easily producing reusable code. Not even object orientation succeeds, despite all claims on its advantages concerning reusability.

By examining existing paradigms and software architectures, we isolate what traits that are desirable, and what traits are harmful in terms of providing means of reusability. Based on these findings we present a new object-based programming model that facilitates reusability in code and concepts through implicit invocation, context dependency, inheritance and polymorphism.

The proposed blob-oriented model does away with properties that hinder production of reusable code, such as coupling and message passing. By removing such traits, the model also provides better means for reusability and incidentally also means for implicit concurrency.

Acknowledgements

First and foremost we would like to extend our deepest gratitude to our supervisor Dr. Tobias Wrigstad for his transatlantic commitment and support. His expert guidance and always spot-on advise makes Tobias nothing short of a hero to us.

We would also very much like to thank Beatrice Åkerblom, Henrik Bergström, Mikael Kindborg and Jozef Swiatycki for patiently listening to our endless rants about blobs and blobjects. We hereby return to you your ears.

Contents

List of Figures

1 Introduction

REUSABILITY HAS LONG SINCE BEEN THE HOLY GRAIL of programming because while hardware components keep decreasing in price, the cost of software development does not [67]. By increasing the degree of reuse in a system, cost and time spent on a project can be dramatically decreased [30, 60, 12]. Also, a higher degree of reuse leads to improvement in system quality and eases software maintenance [12, 56].

In order to find a way of increasing reusability in software development we examine existing programming paradigms and architectures, specifically observing their interplay with reusability. The traits found desirable in order to achieve a higher degree of reuse are identified and combined to form a new programming model that facilitates reuse of code and concepts. The *blob-oriented model* is presented and described as an object-oriented general purpose programming model with extensive means of producing reusable code.

1.1 Background

Since the rise of higher level programming languages, starting with the introduction of Fortran in 1956, numerous alternative ways of writing code and composing systems have been presented [63]. Eder et. al. [28] identifies the driving force behind this development as the strive for producing quality systems recognisable by their extendability, understandability, maintainability and reusability. Boldyreff [12], Frakes and Terry [30], Rockley et. al. [60] and Card et. al. [18] all agree that great degrees of reusability is highly desirable as it raises systems quality, while at the same time reducing development costs.

According to Sethi [63] and Clark and Wilson [21], the early imperative languages brought reusability by use of modules that enables abstraction through encapsulation and grouping of concepts. Declarative languages brings some degree of reusability through functions, but as concluded by Succi et. al. [70], functional and logical programming languages lack many of the features considered to facilitate production of reusable code. While object orientation supports reuse through parametric polymorphism, function overloading and sub-typing [47], Biddle et. al. [11] and Doublait et. al. [27] both stress that reusable code does not come automatically just by using the object-oriented model, but must always be explicitly planned for.

In addition to programming paradigms and models, there are software architectures that are said to promote reusability, such as blackboards and layered systems. However, Bosch [13] and Batory and O'Malley [8] claim that the only real benefit with such systems is that they encourage loose coupling between components.

We believe that non-system specific code should be effortlessly reusable, and in this thesis we propose the blob-oriented model, a programming model that will help enforce production of reusable code.

1.2 The Blob-Oriented Model

Blob-orientation is proposed as a model that facilitates creation of reusable code and concepts in software development, partly by abolishing explicit message passing and external coupling through referencing. The model is object-based, and introduces two views of an object—a generic view and a system-specific view. The generic view is called a *blobject* and the system-specific view is called *a blob*. The blobject constitutes the structure, i.e. variables and internal operations, of an object and contains no knowledge of the outside world. The blob is the part that contains behaviour for and knowledge about blobjects.

Communication in the blob-oriented model is not explicit. Instead of explicit message passing, as is common in object-oriented models, the blob-oriented model handles communication between blobjects implicitly, much in the same way event-based models work. As blobjects are free from knowledge about anything but themselves they can not contain references to other blobjects, thus only handle primitive data types. This limits the implicit communication between blobjects to include only strings, integers and booleans.

The blobject code describes the structure of an object. It contains member variables and procedures that can operate on these variables. The blob can contain any number of blobjects, and operate on them through the use of rule sets. A rule set is a set of conditions and expressions that specify a certain situation and what operations to perform when that situation occurs. This means that a blob will dictate how the blobjects that it contains behave, and what behaviour they use in a specific situation, thus making the behaviour of a blobject context-dependent. The blob in which a blobject currently resides constitutes the current context of the blobject.

For a blob to assign behaviour to a blobject, the blobject needs to reside in the blob. By forming programs by putting blobjects inside blobs, applications are constructed as nested hierarchies of layers. Communication in these hierarchies can only be done between neighbouring layers, disallowing forks in a hierarchy to directly communicate with each other. This allows forks to execute in parallel, independent of each other.

Through the use of hierarchies, blobs can ”dynamically inherit” rule sets from other blobs higher up in the hierarchy, enabling dynamic, context-dependent changes in the behaviour of any blobject. All this is explained in detail in this thesis.

1.3 Methodology

We have used a heuristic approach when working with this thesis. Our conclusions are derived from the literature we have studied, and from the iterative exploratory design of our programming model.

We identify what properties in current programming models that facilitate code reuse, and what properties cause problems with reusability. By identifying properties that help enforce reusability, we use them as stepping-stones to creating a new programming model in which we avoid incorporating any of the the malicious properties we uncover.

The reusability properties in the blob-oriented model are evaluated using three software problems, the first based on data composition, the second on algorithmic expressiveness and the final one based on component flexibility. Each problem is solved using the blob model and already existing paradigms and architectures. The solution of each method is evaluated against three qualitative reusability measurments defined in Chapter 3.

1.4 Problem Statement

Reusability increases both productivity and quality of software in construction [30]. However, writing reusable code is made difficult by the need for explicit design for reuse [11, 27, 67]. How can we design a programming model such that it produces readily reusable code, without imposing demands for explicit design for reuse upon the programmer?

1.5 Purpose

The purpose of this thesis is to facilitate producing reusable code without constantly having to consider reuse during the course of design and development.

1.6 Goal

The goal of this thesis is to present an approach to programming that facilitate building reusable software, without the need for explicit design for reuse. In order to achieve this, the model needs to cope with composition of separately developed, non-interdependent components, and seamlessly handle introduction of new functionality to an already existing system. Furthermore, the model needs to facilitate alternations to, or removal of, existing components. Components should be independent enough to move from one system to another without making any major alterations to it. Last, the model should preserve desirable object-oriented features, such as encapsulation, polymorphism and inheritance, while at the same time discarding less desirable side-effects, such as coupling and other inter-dependencies.

1.7 Delimitations

We analyse the major academical and industrial programming paradigms, as well as some more or less bleeding edge technologies that relate to our work. The software architectures we explore are the ones found to be of value to our programming model because of their reusability claims.

No implementation of the blob-oriented model is presented, and as the model is exploratory and needs further practical evaluation, formalising it was deemed unnecessary. Instead, we focus on exploring and describing the theories behind the model, though sometimes in a very practical manner.

1.8 Outline

In the upcoming chapter we examine some of the existing programming paradigms and architectures and relate them to both reusability and to the blob-oriented model. In Chapter three we define three measurements of reuse and discuss how some of the programming properties found in Chapter two affect these measures. The blob-oriented model is presented in depth in Chapter four. In Chapter five we evaluate the blob-oriented model by using it and some of the architectures and paradigms described in Chapter two to solve three programming problems. The solutions are then evaluated using the measurements described in Chapter three. Chapter six concludes our thesis, we present critique on the blob and outline future research directions.

2

Background and Related Work

In this chapter we review some of the existing programming methods and take extra care to examine their relations to software reusability. By doing this we identify properties that affect reusability, and use the knowledge gained in this chapter to help construct the blob-oriented model. First, though, we elaborate on software reusability in order to create a clearer understanding of the concept.

2.1 Reusability

In order to identify what reusability really is, this section describes means of reusability in software development and explains a couple of common reuse concepts. Frakes and Terry [30] claim that software reuse is applicable to all stages of the software development cycle, ranging from cost estimation and requirements specification to system design and source code. However, the concern of this thesis is primarily reuse of concepts, source code and components, and thus only these aspects of reusability are covered. In order to pinpoint what reusability aspects provide the highest degrees of reuse, a short evaluation of the different methods of reuse concludes this section.

2.1.1 Software Reuse

A technical report undertaken for the Enacts Network regarding reusability in software [56] concludes that incorporating reusability in software has several advantages, two being increased software productivity and software quality. This claim is supported by Frakes and Terry [30], Rockley et. al. [60] and Boldyreff [12]. They all agree that reusability decreases cost and time spent on software through minimising the need for maintenance and modifications to software. A study by Card et. al. [19] shows that the cost of reusing a line of code is merely 20 percent of the cost of writing a new line.

Reuse of Source Code

Doublait [27] describes code reuse as being reuse of bits of source code without the related documentation. The Enacts report [56] calls such code reuse *unplanned reuse*, and suggests that though it might be quick, it might also be counterproductive in producing high-quality software. Instead, the Enacts report [56] propose that *planned reuse*, a process where bits of code is reused and also thoroughly tested and documented is a better, though more costly method of reusing code.

Reuse of Components and Modules

Szyperski [71] describes a component as an independent software unit which can be grouped with other units to form a functioning system. In order to create reusable components, Szyperski claims that one must focus on implementing only the core functionality of a component, and outsource any other behaviour to other components rather than implementing it in the specific module. This way components are made less redundant, more concentrated on doing their job and nothing but their job, and are also forced to work together, thus theoretically enhancing the reusability of the component itself [18, 71].

The way components are built for reuse is closely related to the way modules and software libraries are constructed [56]. A study conducted by Card et. al. [18] on software reuse in modules showed that small, single-function modules exhibited the highest reusability factor, and that these small modules would often offer reusability without making any modifications to them.

Reuse of Concepts

According to Boldyreff [12], the reuse of concepts is the key to recognising the opportunity for reuse in existing software. By cataloguing concepts, as is done in a software component library, components are made searchable and findable to application programmers. Cataloguing of concepts becomes easier as abstraction levels increase, as higher abstraction leads to clearer and more modularised concepts [12].

Revenue from Reusability Methods

In the technical report on software reusability produced for the Enacts Network [56] it is concluded that system design and architecture is the key to creating reusable software. Doublait [27] supports this claim and explains that producing source code represents less than a quarter of the total development cost in software production. This indicates that reuse of source code should perhaps not be the prime concern when targeting reusability. Though, the necessity for code to be prepared for reuse is emphasised by the Enacts report [56], Sommerville [67], Doublait [27] and Johnson and Foote [39], as readily reusable source code has immediate effect on productivity and the extent to which software can be reused. Doublait also claims that reusability requirements should be made on a high level of abstraction so that reuse of components and source code is anticipated [27], thus allowing the concept of reuse to pervade all levels of the software development process. These findings lead to the conclusion that focusing on reuse of concepts provides more revenue than reuse of components and source code alone.

2.2 Programming Paradigms

In this section we investigate characteristics of some more or less common programming paradigms and concepts in order to identify their means of reusability. The concepts we examine are programming models that capture different semantics for constructing programs and writing code.

2.2.1 Imperative Languages

The imperative language family dates back to the introduction of Fortran in 1956[1], and includes languages such as C, Pascal, Algol and Cobol [63]. When introduced, Fortran brought features like subprograms with parameters, formatted i/o and abstract data types, thus raising abstraction levels from its machine language predecessors [22, 26, 76]. The main use of Fortran, and later also Algol-60, was numerical computations, while Cobol was used for commercial data processing. Imperative languages has since evolved and constituted the most common programming paradigm, at least in industrial settings [76].

Program execution in imperative languages is managed in a procedural fashion. Through sequences of actions, such as procedure calls or variable assignments [6], machine state is altered by moving bits of data around. Because of this step-wise execution imperative languages can be implemented effectively, at least in theory, as imperative languages and hardware handle data in the same fashion [6, 76].

In order to produce reusable code in imperative languages, functions are an important tool as they facilitate grouping collections of instructions under a single name. In imperative languages functions can exist on their own and provide significant abstraction that lets programmers write code once and then use it by calling the function from other parts of the program without having to duplicate code [22, 63, 76].

[1] The manual for Fortran was introduced in 1956, but there was no working compiler until 1958 [22].

A higher degree of reusability can be achieved through use of modules that enable data hiding—protecting variables from being updated from any part of the program. Data hiding also facilitates separation of responsibility and helps with module encapsulation [6, 56, 76]. Modules may be used to simulate types, grouping data and operations upon that data together [63]. In order to be considered reusable, a module should define a set of operations through a small interface while hiding the implementation from the user programmer, thus raising the level of abstraction [56, 22]. The interface defined needs to be standardised enough to make it easy to ”glue” components together in order to produce larger pieces of software [56].

The blob-oriented model uses procedural algorithms to describe behaviour in rule sets, and also in procedures that describe operations in blobjects. Though the model can be used to create modules, there is no need for a blob-oriented module to implement any interfaces as communication and behaviour is dictated by the current execution context.

2.2.2 Declarative Languages

Declarative languages mainly belong to the functional and logical programming paradigms, and earn their common definition from the way code is written [6]. Instead of procedural sequences and assignments as in imperative languages, code is constructed as declarative statements and expressions. This allows a programmer to produce code at a higher level of abstraction than in the less expressive imperative languages.

One of the major advantages that declarative languages are considered to have over imperative languages is the notion that the code produced is more understandable [6, 59]. Backus supported this claim during his lecture [5] at the 1977 *Turing Awards*, and meant that this is thanks to the fact that the semantics of expressions in declarative languages is independent of its textual or run-time context, as opposed to imperative languages where the semantics is context-dependent. In imperative languages one needs to be familiar with everything that might affect a statement during execution or compilation, such as scoping rules or tracing message passing and variable assignments, in order to determine the semantics of a statement [59]. Thus, declarative languages give the programmer the ability to construct programs without explicitly declaring the complete problem-solving process, which is something considered to be a problem when programming computers [59]. Instead of describing the complete execution flow in sequences, a programmer can simply describe rules that capture the semantics of a program. The key feature in declarative languages is the notion that a programmer does not specify how a computation is to be performed, but rather what is to be computed [6]. For example, to calculate the sum of all numbers in a list in an imperative language, one would describe the sequence of going through the list one number at a time adding its value to a variable which holds the accumulated sum.

```
int accumulated_sum = 0;
for (int x = 0; x < list.length(); x++)
    accumulated_sum += list[x];
```

In a declarative language, in this case Prolog, one would not describe the sequence, but rather different states. The first state is the empty list, in which case the sum of the values in the list is 0. If the list is not empty the value of the first element in the list should be added to the accumulated sum. The element will then be removed, and the procedure will be repeated.

```
1 sum([], 0).
2 sum([First|Rest], AccumulatedSum) :-
3     sum(Rest, Temp), AccumulatedSum is Temp + First.
```

By use of polymorphism to overload the rule `sum`, Prolog uses the current state of the program to choose the next execution path. The rule on line 1 says that if the list is empty, the accumulated sum is 0. The rule on line 2 says that if the list is not empty, the accumulated sum is the sum of the value of the first element in the list and the accumulated sum of the same calculation using the next element in the list.

It is usually more difficult to make a program written in a declarative language as efficient as one written in an imperative language [6], as declarative languages are not as machine-centered as imperative languages. Also, declarative languages generally provide few or no means of producing reusable code.

Through the use of rule sets, a blob implements constructs similar to those of declarative languages where polymorphic rules are applied to the current state of the program. Depending on the set up of the current context in a blob, different rules will be executed.

Functional Programming

Functional languages stem from the idea of describing computer programs as mathematical functions [6, 59, 63]. The building-blocks of such languages are functions and expressions mapped to functions.

Functional programming began with the language Lisp, invented by John McCarthy in 1958 for use in artificial intelligence [63]. Lisp was designed to aid in the creation of a system that would use deduction to solve problems, and thus exhibit

”common sense” [51]. Lisp then evolved into a simpler model, a formal mathematical language where all data was represented as symbolic expressions, and came to be designed primarily for symbolic data processing [52].

In a purely functional language, the order of execution or evaluation is not decided by sequences or iterative repetitions, but rather conditions and recursion [59]. A function or an expression always produces the same result given the same input, which results in that no instructions in purely functional languages cause side effects[2] [59].

Though functional languages provide no revolutionary means of reuse, Wadler [75] claims they do support reuse through use of modules. Wadler also claims that the built-in data structures in functional languages eliminate the risk of unwanted side-effects, and also benefit backward compatibility. This, Wadler continues, is desirable from a reusability point of view.

Logic Programming

In the late 1960’s and early 1970’s research was conducted on automatic resolution inference [21], meaning the ability to draw conclusions from known facts using an automated system. In the introduction to their collection of papers regarding such systems, Clark and Tärnlund [21] describe how the rapid development in this field would result in a language called Prolog, and with Prolog a new programming paradigm—*logic programming*.

Prolog was, at first, a language built to process natural language, but was found to have all the qualities necessary for solving the same kinds of problems as languages such as Lisp [63]. Logic programming languages use facts and rules to represent information and algorithms, and use deduction to answer questions, or *queries* [63]. For example, the following Prolog program uses deduction to figure out if apples are tasty, based on the facts declared in the program.

```
fruit(apple).
tasty(Thing) :- fruit(Thing).
```

By knowing that an apple is a fruit, and that all fruits are tasty, the program can conclude that apples are tasty given the query `tasty(apple)`.

The *logic* in logic programming lies within the way a program is structured, which is describing the logical structure of a problem rather than describing to a computer exactly what it needs to do in order to solve a problem [37, 59]. Kowalski [43] describes this way of programming as a paradigm that makes use of the fact that

[2] The fact that an instruction does more than it is supposed to do, which can cause unexpected behaviour in unforeseen parts of a program.

one can express computable procedures and functions using logic. It also makes use of goal-directed deductive methods of proving statements in order to run these expressions as a computer program.

The use of a declarative language and deductive reasoning, as opposed to imperative languages algorithmic structure, allowed for a raised level of abstraction in problem declaration and the method of execution. With the use of declarative logic statements, also known as *propositions*, the semantics of a given statement is much easier to determine than a declaration in an imperative language, as in all declarative languages [6].

Downsides to this paradigm include the problems with modularisation and code reusability, as identified by Succi et. al. in their study of reusability in logical programming [70]. They conclude that—even though logical languages include properties shared with functional languages such as absence of side effects and ripple effects[3], potential for defining patterns and potential for better understandability than imperative languages—this paradigm does not offer any real means of data abstraction or information hiding.

2.2.3 Object Orientation

The concepts of classes and objects in programming was introduced in 1963 through the programming language Simula [63]. Simula was originally built for performing computer simulations, but was later revised to manage less specific types of computation [63]. The term *object-oriented*, however, was not used until the development of Smalltalk in 1972 [17].

In 1980, Stroustrup began his work on extending C with access control, constructors, classes and inheritance, a work that resulted in the language C with Classes. By 1984 dynamic binding, overloading and reference types was introduced through virtual methods, an inclusion that resulted in C^{++} [59]. C^{++} brought object orientation widespread acceptance in industrial settings during the 1990's [26], mainly due to inexpensive and available compilers, its (almost) backward compatibility with C, and the use of static type checking [59].

Object orientation was presented as a model built to provide good support for reuse of code, concepts and models, mainly through use of encapsulation, polymorphism, generics and inheritance [16, 56, 58, 47]. However, despite providing tools for reusability, claims that object orientation has failed to deliver in terms of reuse have been made [31, 73]. The main reason for this, says Biddle et. al. [11], is that object orientation has been presented as something simple and natural, which has lead programmers to believe that they do not need to put any effort into making

[3] The fact that a change in one part of a program changes the behaviour in other parts of a program.

code reusable, but will instead just expect reusability to happen. Better understanding of the technique, they argue, would make for better use of object orientation in terms of reuse. The thought of object-oriented thinking not being that ”natural” fits in well with Baniassad and Fleisners’s [7] findings. After having conducted a study on how easterners and westerners perceive a typically object-oriented scene, Baniassad and Fleisner conclude that the object-oriented view does not map that well to how easterners perceive the world. It seamed easterners were more prone to identify interaction and general compositions of objects, rather than isolating single objects and assigning properties and behaviour to them. Doublait [27] and Biddle [11] agree that reuse is nothing purely intuitive, but rather something that programmers need to put effort and work into in order to achieve in an object-oriented setting. Lewis et. al. [47] agrees with this notion and states that software must be designed for reuse in order to be reusable. Doublait [27] also concludes that reuse is easier to achieve on a smaller scale, where programmers are already familiar with the implementation details of a component, rather than on a greater level, where programmers may not know much about the component they are trying to reuse. This goes directly against the claim that encapsulation strengthens reuse levels [16, 56, 47, 58], as Doublait’s findings mean that a component first needs to be disassembled and examined before it can be properly reused.

Sandhu et. al. [61] isolates coupling as one of the major obstacles for achieving reusable object-oriented code, a claim supported by Eder et. al. [28]. As coupling increases, understandability and reusability decreases to a point where the object-oriented code is no longer reusable [61]. Briand et. al. [14] find that increased coupling brings an increased tendency toward software errors.

The blob-oriented model is object-based and thus shares a lot of properties with the object-oriented model—objects, type polymorphism, inheritance and private data to name a few. The big difference, though, is the fact that blobjects have no way of referencing other blobjects, and thus there is no external coupling through referencing nor any explicit message passing. Explicit communication between blobjects has instead been replaced with alternative, implicit communication styles as discussed later on in this chapter.

2.2.4 Extensions to Object Orientation

There are numerous extensions to object orientation that try and fix the problems described in Section 2.2.3 by extending the object oriented model. We examine two of these extensions, *aspect-oriented programming* and *context-oriented programming* in order to see what solutions they offer.

Aspect-Oriented Programming

Aspect-oriented programming was first introduced by Kiczales et. al. [40] in 1997, and aims to solve the problem of cross-cutting concerns in object orientation. At the base of aspect-oriented programming is the object-oriented model. All programs are structured as correlations of interrelated objects that are all instances of some class which is a part of a class hierarchy [41]. Kiczales et. al. [40] argue that when trying to model cross-cutting concerns in an object-oriented setting it results in code tangling, where small amounts of domain specific code is scattered all over a program. The reason for this, claims Shukla et. al. [65], is that programmers are trained to make classes out of all nouns, such as ”account” and ”customer”, and thus do so for things that are not necessarily best modeled as objects, such as ”logger” or ”printer”. The problems discussed by Kiczales et. al. [41] are solved by turning cross-cutting concerns into *aspects*, and thereby moving them into separate modules. An aspect is a construct containing code relating to one system concern, for example thread safety or logging [41]. The aspects works by recognising patterns in program execution and run related code whenever conditions for the aspect is met [40]. This approach to systems modeling enables programmers to write cleaner code with less code tangling. Such code is easier to reuse as it is more modularised and includes less inter-dependencies. [55, 65].

In contrast, Steimann [68] expresses his concern over the monotonous examples of what aspects can do, i.e. logging, tracing and debugging, and conjectures that most of the issues named by proponents of the paradigm would be better solved by an intelligent IDE or by language extensions. Further on, Steimann concludes that aspects in fact break modularity rather than increasing it, while at the same time reducing readability by introducing obliviousness.

The concept of aspects has since its introduction been implemented in numerous languages from different paradigms, thus allowing separation of cross-cutting concerns in other settings than only object orientation.

The blob-oriented model shares some properties with aspects, mainly that there is code placed outside a blobject that affects its behaviour while the blobject remains oblivious to this. Though this approach might reduce readability of the code, we argue that combining aspects with context-dependencies will provide raised cohesion as we discuss in the upcoming section.

Context-Oriented Programming

Context-oriented programming was first implemented by Costanza and Hirschfeld [24] as a language extension of the Common Lisp Object System. Context-oriented programming was designed to handle problems arising in object-oriented settings when implementing context dependent behaviour. Such problems arise when a system needs to have multiple views of an object, an occurrence that, according to Costanza and Hirschfeld [25], leads to separation and scattering of code. This may be exemplified using the *Model-View-Controller* framework [44], where code for viewing an object is contained within one layer, code for modifying an object is kept in a model layer, and code for managing synchronisation between the model and the view is kept in a controller layer.

According to Hirschfeld et. al. [35], a context-oriented program does away with these issues through use of layers which allow for multiple views on a single object. Layers are first-class entities in context-oriented programming, and are used to group context-dependent behaviour together. This approach makes it possible for an object to produce different behaviour depending on its active layers. Hirschfeld et. al. [35] emphasise the positive gains for reusability in having objects behave differently depending on their context, as such a set up allows for keeping all object code within a single object, rather than scattering it all across the program.

We agree with Costanza and Hirshfeld [25] that the behaviour of an object should depend on its current context. Though we do not agree that the object itself should contain the context-dependent behaviour as this causes objects to be bloated with context knowledge. Blobjects are context-dependent, thus their behaviour is specified in the context rather than in the blobject, creating a kind of context-dependent aspect. This way blobjects are free from knowledge about the contexts they can function in, which allows them to function in any context that handles the blobject. By treating the context as the container for context-specific behaviour, blobjects are left clean and overall system cohesion is raised.

2.2.5 Event-Driven Programming

The event-driven programming paradigm distinguishes itself from other paradigms in the way a program executes. Event-driven programs react to events during execution, rather than performing a controlled, structured task from start to goal [72]. This means that there is no telling in which order execution will occur—it is dependent on the input a program receives and when this input is received.

Event-driven programming is also called *interaction-based* programming, as event-driven applications receive input from other entities that interact with the program [78]. Interaction is rarely linear, but rather unpredictable both when it comes to timing and data. This means that event-driven programs must be prepared to handle interaction at any given moment, which brings us to one other characteristics of this paradigm—interaction-based programs are designed to run for an arbitrary period of time [72]. In contrast, programs in other paradigms usually have a well-defined control sequence which will terminate a program once an execution path from start to goal has been run.

Event-based interaction is built up around components that may or may not trigger on a given event during run-time [64, 72]. This property provides strong reusability support, as components can be installed or upgraded without affecting other components in the system [64].

Wegner [78] claims that interaction-based programming is much more powerful than traditional algorithmic paradigms. Wegner, with support from Milner [53] and others [46, 49], says that interaction cannot be expressed solely by the use of algorithms, and thus interactive software extends beyond algorithmic and mathematical formalisation which, according to Wegner [78], are causes of restraint of expressiveness in programming languages.

The most common problems with event-based systems are related to execution and data [64]. In this type of systems, it's hard to gain control over execution order. There are also problems with shared data between modules, as modules do not communicate explicitly with each other.

The blob model uses event-like communication between blobjects, where primitive values are implicitly distributed to all blobjects that are currently interested in reacting to or using input. As discussed in Section 4.1, the model has a predictable control flow where the primitive values are distributed in a pre-defined sequence, allowing the programmer to foresee execution flow.

2.2.6 Concurrent Programming

Concurrency in a programming language refers to the potential of parallel execution of a program, meaning the ability to execute two or more processes in parallel [63]. Concurrent programming as an abstraction deal with *abstract* parallelism [9], meaning that the actual properties of the parallel execution—whether the execution is distributed over several physical processors or whether the concurrency is simply ”mimicked” by one single processor—remain insignificant to the programmer.

Execution of parallel processes is asynchronous [2, 9], and as parallel processes execute independently from one another, their respective execution rate is never static. This puts stress on the correctness of a program when states or variables are shared between processes [9, 34], as a process can never assume that another process has or has not changed the state or variable, or is not currently using this state or variable in a computation. Thus, there is a need for parallel processes to be able to communicate and synchronise [9].

Apart from shared data, parallel processes can use communication, or *explicit message passing* [9, 4] to exchange information. Message passing can be both synchronous and asynchronous, where synchronous requires both the sender and receiver of a message to be prepared and ready to exchange information. If the receiver is not able to process the message when it is sent, the sending process will block until the receiving process is able to handle it. Asynchronous message passing is when the sender does not wait for confirmation from the receiver when a message is sent, and thus does not know when the receiver will process the message [9].

As the blob-oriented model does not allow parallel processes to communicate, concurrency in the blob model does not need to be explicit. As there is no shared data nor message passing, concurrent processes can not affect each other and thus have no need to synchronise.

The Join-Calculus

The join-calculus, developed by Fournet and Gonthier [29], is a basic model of concurrency that allows for parallelism in typical object-oriented languages, thus introducing the possibility of real language constructs for concurrency instead of having to rely on libraries to achieve parallelism. The basic idea is that a non-concurrent higher-order language can be extended with join patterns and fork calls in order to offer concurrency and synchronisation [29]. This feature will give programmers a higher level of abstraction to work with than using explicit threading with message passing for concurrency and synchronisation [10].

A join pattern allows a programmer to specify a set of two or more method headers that define what events need to take place before executing a method body. When all methods in the header have been called, the join pattern is complete and the method body can be executed [10].

Though the join-calculus allows for higher levels of abstraction than threading when dealing with concurrency, the use of concurrency mechanisms is still explicitly declared by the programmer [10, 29, 38, 48].

There is a join-calculus extension to $C^{\#}$ developed by Benton and Fournet [10]. Their approach, called *Polyphonic $C^{\#}$*[4], contains ideas similar to those of event-based systems and is meant to facilitate asynchronous concurrency in $C^{\#}$. Benton and Fournet [10] argue that asynchronous events are increasingly common on all levels in software systems, and that there is a need for real language constructs that address asynchrony and concurrency, rather than using concurrency-enabling libraries such as threads. There are similar extensions to other object-oriented languages, such as Join Java [38] and JoCaml [48], that make use of the join-calculus in order to implement constructs for concurrency in Java and OCaml.

[4] Polyphonic $C^{\#}$ is now integrated with the $C^{\#}$ extension *$C\omega$*.

The blob model uses join patterns in the system-specific rule sets, where the rule headers define a number of conditions that need to be met in order for a rule to execute. The major difference between the use of join patterns in the blob-oriented model and in the join calculus is that activation of the rule headers in a blob is implicit and always dependent on the current context.

2.3 Architectures

In this section, we review two software architectures with claims on reusability and pluggability in order to obtain a higher-level view on reusability in software engineering. These architectures are independent from programming paradigms, and instead suggest how to structure a system of programs or components.

2.3.1 Blackboard Systems

A software architecture loosely related to event-driven programming is the *blackboard* architecture. According to Shaw and Garlan [64], blackboards and event-based systems share the fact that the order in which components execute is not explicitly declared, but instead dependent on the current available input which a component is willing to accept.

Bosch [13] explains that a blackboard system consists of a data repository, the blackboard, and components that more or less actively reap data from the repository (see Figure 2.1). A component scans the blackboard for input that the component is able to process, grabs it, processes it and then puts the result back up on the blackboard, making it available for other components to process. This kind of architecture originates from the AI field, where it was developed for facilitating speech recognition, pattern-matching and similar tasks [13, 64]. Blackboards are also used in systems where components with loose coupling need to operate on a common data pool.

According to Bosch [13] the structure of a blackboard system allows for high reuse. As this kind of system consists of a data repository and any number of independent components communicating with the repository, both the data types and the components are interchangeable [13]. Components can even be arbitrarily plugged or unplugged from the system, as they do not explicitly affect each other.

One common problem with the blackboard architecture, says Bosch [13], is performance. As the execution order is not explicitly described, computations on data from the backboard might not be performed in an optimal order. However, Corkill [23] claims that blackboard systems do not need to be slow. Corkill means that this misconception stems from the early blackboard systems that had to be built from scratch for each new application. The implementer's understanding of such

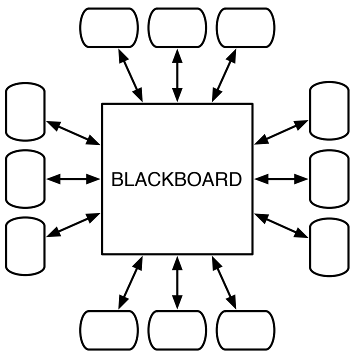


Fig. 2.1. A Blackboard System. Components read data from the shared memory space, performs operations on it, and then writes the data back onto the blackboard.

a system had to be based on higher-level descriptions of blackboards that lacked implementation details, thus making it difficult to implement an optimised system. Corkill [23] exemplifies a number of more recent blackboard frameworks successfully deployed, and claims they do not suffer from the poor performance common in the early blackboard systems. Blackboard systems might also suffer from problems with reliability and safety as there is no explicit control structure that makes sure the system behaves in an expected manner [13].

The blob model uses a blackboard-like construct in the area where blobjects are contained by a blob. Instead of components interacting with a data repository, as is the case with the blackboard architecture, the blob-oriented model allows the current context to describe how components interact. This preserves the benefits of components being loosely coupled and dynamically interchangeable.

2.3.2 Layered Systems

The concept of layered systems is based on the idea that components are arranged in a hierarchy where one component, or a *layer*, only interacts with its neighbouring layers [64]. This structure gives layered systems properties that allow systems to be built in a hierarchy where the level of abstraction increases with each layer. According to Shaw and Garlan [64] such systems are advantageous when it comes to reuse, as components at most communicate with only two other layers, and thus are not heavily tangled with knowledge about and/or references to other, more distant parts of the system. For the same reasons, layered systems also facilitate interchangeability of components [8, 64], as only interfaces from at most two other layers must be implemented.

Szyperski [71] describes two different kinds of layered systems, *strict layering* and *non-strict layering*. In the strictly layered system, seen in Figure 2.2, a layer may only have access to layers situated immediately below itself, whereas the non-strict layering allows a layer to access any of the lower layers [71]. One problem with strictly layered systems, according to Szyperski [71], is issues with extensibility. Such issues arise as every new layer can only implement behaviour supported by a previous layer, which means that even though a desired service may be available in some lower layer or in the hardware, it is made inaccessible by some layer above it. This, Szyperski [71] says, may be remedied by having each layer allow extensions, so that the extension can be placed in the layer where it can most easily access the service it needs.

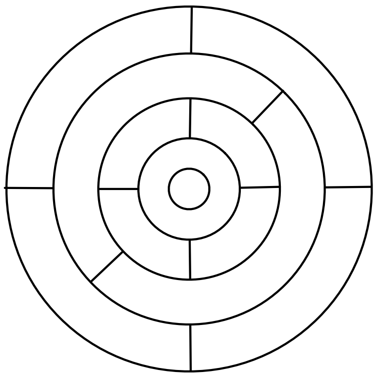


Fig. 2.2. A Strictly Layered System. Each layer may only access the services of the layer or layers directly below it.

Shaw and Garlan [64] describe further issues that arise when using hierarchically structured systems, for instance when a system requires interaction to bridge several layers or when a domain is not intuitively broken down into layers of increasing abstractions. There can also be extensibility issues, as layers with higher levels of abstraction are built on top of a previous layer, which limits the amount of increased abstraction one additional layer can supply [71]. Also, far from all systems can be decomposed into layers, which disqualifies this architecture from being generally applicable.

The blob-oriented model enforces a strictly layered architecture, as it is described by Szyperski [71], throughout the system by the use of hierarchies where each layer has access only to the layers directly below it. As blobjects are free to move around, they may be placed in a layer where they can access services needed, which also makes the system extensible. This provides scalability across all levels of a system, from a single blobject to blob large modules, as it enables an increasing level of abstraction in the same way layered systems do. Because the blob model is free from external coupling through referencing, pluggability is even leaner in the blob model than in layered systems as there is no need for a layer to implement any interfaces to neighbouring layers, but only—though optional—behaviour for operating on throughput. The blob-oriented model is also better at handling interaction that bridges several layers as the rule sets describing interaction propagate downwards through the hierarchies, thus creating a way for behaviour to travel across layers.

2.4 Concluding Remarks

In this chapter we have examined the concept of reuse and elaborated on reusability properties. We have also reviewed some of the existing programming paradigms and software architectures, and discussed their relation to both reusability and the blob-oriented model. All paradigms and architectures described in this chapter facilitate the possibility for reuse, even if none have managed to completely solve the reusability problem. Also, many of the existing ideas on how to achieve reusability seem to be generally good, and should therefore not be tossed aside. In the following chapter we further isolate these gems of reusability in order to establish what properties of a programming model that could help make software reuse easier.

❖ 3

Reusability Properties Examined

In this chapter we further examine some of the properties identified in the programming paradigms and architectures described in the previous chapter. These properties all relate to reusability by either enforcing or decreasing it. In order to determine how desirable a property is, we define three qualitative measurements of reusability and apply them to the properties.

3.1 Measurements of Reusability

In this section we describe coupling, cohesion and abstraction as three reusability measurements. These properties are used throughout this thesis to reason about reuse, and are also used to evaluate how well the blob-oriented model supports reusability.

3.1.1 Coupling

In his work on software measurements regarding coupling, Alghamdi [1] names coupling as a very important factor in terms of achieving software quality. Highly coupled components make for bad pluggability and exchangeability, which leads to less maintainable systems. It is, says Alghamdi, desirable to strive for as low coupling as possible when constructing a system, a claim that is supported by Eder et. al. [28], Stevens et. al. [69] and Sandhu et. al. [61].

Eder et. al. [28] define three types of coupling specific for object-oriented systems—interaction, inheritance and component coupling. The first one, interaction coupling, implies the coupling between methods or data slots that may or may not belong to the same class. In the code below the method `getAddress()` is directly dependent on the method `getPostalCode()` and the instance variable `street`, making them internally coupled through interaction.

```
getAddress(){
   return street + getPostalCode();
}

getPostalCode(){ ... }
```

Interaction coupling, says Eder et. al. [28], is the form of coupling most intimately related to the module coupling that occurs in procedural programming. Object orientation has done away with some of the module coupling, described by Stevens et. al. [69] as the procedural practice of sharing a global data space among modules, by introducing more efficient data hiding through access modifiers and encapsulation [28]. Interaction coupling, continues Eder et. al., may be both internal and external. External coupling occurs whenever a separate class accesses data slots or methods in another class, the worst form of which, according to Eder et. al., is the passing of public instance variables between two methods belonging to a different class. An example of this may be seen in the code below, where the application code directly accesses the variable `name` in the object `Person`, and sends it as an argument to a method in a modifier class.

```
class Person{
   public String name;
}

class Modifier{
   modifyString( String aString ) {
      aString.reverse();
   }
}

class Application{
   p = new Person();
   mod = new Modifier();
   mod.modifyString( p.name );
}
```

Internal coupling occurs whenever methods residing in a class are dependent upon non-transient[1] methods or instance variables within that same class. This kind of coupling is not to be considered harmful as long as the occurrence is separate from inheritance, in which case the coupling is to be rated inherited external, and is thus

[1] The term non-transient refers to data dealing with inner state of an object.

much more severe [28]. An example of this type of coupling is shown below, where a class `Person` implements an access method for retrieving the instance variable `name`.

```
class Person{
   private String name;

   getName() {
      return name;
   }
}
```

The second form of object-oriented coupling described by Eder et. al. [28] is inheritance coupling. As previously discussed, inheritance coupling occurs whenever a derived method is dependent upon instance variables or methods residing in a super class. Eder et. al. describe this form of coupling as being severe whenever a derived class changes the interface signature of the parent class, i.e. by removing inherited methods or overriding behaviour in such a way that it changes the semantics of the derived method. A trivial example is shown in the code below, where the subclass `Employee` overrides the `getName` method declared in the `Person` class, and changing its semantics to not return the name of the person, but to instead return a title. The example also shows inheritance coupling where the employee class alters the instance variable `name` declared in the parent class.

```
class Person{
   String name;

   getName(){
      return name;
   }
}

class Employee extends Person{
   getName(){
      name = '';
      return title;
   }
}
```

As mentioned earlier, it is desirable to strive for as low coupling as possible when constructing a system in order to achieve higher software quality and reusability [28, 61, 69]. However, some forms of inheritance coupling will, contradictory enough, coincide with better reusability, as an application class handling an object of some class need only be coupled with that class in order to safely handle any of its derived classes [28].

The last type of object-oriented coupling described by Eder et. al. [28] is component coupling. Component coupling is very much like the previously discussed interaction coupling, but differs in that interaction coupling has to do with the amount of data being shared and the complexity of this data, while component coupling has to do with how explicit the coupling between classes is. The worst form of component coupling, says Eder et. al. [28], is the hidden kind that does not show up in any specification but exists none the less. An example of this is the direct modification of a class obtained by calling a method in another class that may or may not be explicitly coupled with the calling class. The line of code below shows an example of this type of coupling, where the method `setLayout()` is called on an object retrieved by calling the method `getContentPane()` on some instance variable.

```
frame.getContentPane().setLayout(new Layout());
```

This type of coupling, explains Eder et. al. [28], is harmful because specifications are not likely to entail dependencies created in this way, thus making them hard to track.

3.1.2 Cohesion

Schorsch and Cook [62] define cohesion as a way of measuring how many things a module accomplishes. A highly cohesive module, they say, is a module that solves only one problem. It is desirable, from a point of software reuse and quality, to strive for as high cohesion as possible when constructing software as modules with high cohesion are easier to maintain and reuse than modules with low cohesion [62, 28, 69].

As for object-oriented cohesion, Eder et. al. [28] make difference between method, class and inheritance cohesion. Method cohesion, they explain, is the degree to which a method implements behaviour concerning only one thing, rather than just bundling together behaviour that really has nothing in common. Further, the degree of class cohesion depends on how meaningful the composed class is from the point of providing semantically good data abstractions [28]. Finally, Eder et. al. describes inheritance cohesion as the degree of things an inheritance hierarchy is concerned with. Eder et. al. stress the importance of maintaining high cohesion through out the system, as a way of increasing quality and reusability, a notion supported by Schrosch and Cook [62]. A problem in achieving this, they claim, is that programming languages of today provide little support for writing highly cohesive modules. Programmers, they continue, are prone to bundle too many responsibilities into each module, and thus making the module harder to maintain and reuse. Schorsch and Cook do however consider aspect-oriented programming a way of achieving high cohesion, as aspects provide bundling of crosscutting concerns, meaning that such behaviour can be moved from the classes, leaving them cleaner, leaner and more cohesive.

3.1.3 Abstraction

Abstraction is described by Budd [16] and Lewis et. al. [47], among others [56, 58], as an important key to reusability. By being able to view a component or a system in various layers of abstraction, focus can be put on understanding only the parts necessary in order to use the component or system, thus making reuse easier. This claim is supported by Synder [66], Watt [76] and Jipping and Dershen [26].

Abstraction is a broad term that includes encapsulation, inheritance and polymorphism [16], properties regarded as killer features of object-orientation. Encapsulation uses information hiding and brings modularisation, something that enables abstraction in the sense that only the vital details of a component or system are revealed. Inheritance in combination with polymorphism allows for greater generality and more understandable code [16].

Although inheritance is meant to aid abstraction in systems, Weck and Szyperski [77] claim that the way inheritance is used in object-oriented settings today is rather harmful and counterproductive to forming abstraction. They mean that inheritance as subclassing is used only to bypass encapsulation properties assigned to an object, thus corrupting indented abstraction levels.

3.2 Programming Properties

In this section we discuss properties identified in Chapter 2 that affect reusability in software development. We elaborate on their relation to reuse in order to motivate how they are used to increase reusability in the blob-oriented model.

3.2.1 Encapsulation

Snyder [66] defines encapsulation as the restriction of client access to a module via a defined module interface, which allows for changes in the internal implementation of a module without the necessity of changes in surrounding client code. The interface serves as a contract between the component and its users, and ensures that any future reimplementation of the component is backwards compatible with all of its users as long as it does not change the interface. Synder [66], Watt [76] and Jipping and Dershen [26] agree that encapsulation facilitates data abstraction, as application programmers do not need to know about implementation details of a component in order to use it. This, according to Pree [58], Budd [16] and Lewis et.al. [47], leads to greater levels of reuse and provides better maintainability.

As discussed in Section 3.1.3, Weck and Szyperski [77] claim that encapsulation might be broken by the use of inheritance, thus also breaking abstraction. It is the external coupling through inheritance that allow programmers to disobey encapsulation rules defined by a class, thus coupling can be identified as a threat to encapsulation.

The blob-oriented model provides strong encapsulation of blobjects, as described in Section 4.1.2. Data in a blobject can not be modified without the use of a contract, which in the case of blobjects means using member procedures to manipulate state. Though, as blobjects can be inherited from, the problems identified by Weck and Szyperski [77] regarding encapsulation and inheritance coupling could occur in blobjects. The problems with inheritance coupling might show when a blobject class is extended by a new blobject class. If the subclass offers a procedure that manipulates a variable which was not manipulatable in the original blobject class, the original encapsulation is broken.

When it comes to blobs it is difficult to apply the concept of encapsulation to the rule sets, as there is no way for a blob to read, manipulate or extend the rule set of any other blob. Rule sets are thus heavily encapsulated with no means of affecting or accessing each other.

3.2.2 Inheritance

A valuable property in object orientation is inheritance, which facilitates reuse for both code and concepts [16, 26]. It supports reuse of code as it allows programmers to reuse component behaviour through subclassing without rewriting code to suit their present need. Reuse of concepts occur when a child class overrides behaviour in a super class and thus only implements the concept, not the behaviour of the parent class [74, 16]. Biddle et. al. [11] note that the main gain with using inheritance is not that it makes component code reusable but that it, through use of polymorphism, also makes the context code reusable.

As previously discussed, Weck and Szyperski [77] argue that inheritance breaks encapsulation, and thereby also abstraction, to the extent that it should be abolished as a software building practice. As with the case of encapsulation, Weck and Szyperski's [77] claims identify inheritance as a problem when creating reusable code. They argue that overriding breaks concept semantics, and name inheritance coupling as a means of writing programs that are hard to maintain and overlook.

Inheritance in the blob-oriented model is very similar to inheritance in object-orientation. Blobjects can be extended—subclassed and subtyped—and member procedures can be overridden, though not overloaded as procedures in blobjects accept no arguments. Member variables can be accessed and manipulated by subclasses.

As with the case of encapsulation, the blob-oriented model once again faces problems identified by Weck and Szyperski [77] when it comes to inheritance and reusability. By allowing overriding, the semantics of procedures in a blobject taxonomy might vary from class to class, thus making programs more complicated to maintain and overlook.

3.2.3 Polymorphism

Budd [16] explains how polymorphic behaviour in objects is achieved through combining inheritance with overriding of methods. This, says Cardelli [20], helps in keeping interfaces small as not all behaviour is specified within one class, but rather spread over several classes. Snyder [66] and Watt [76] agree that keeping interfaces to a minimum will ease insertion and replacement of classes or modules, thus facilitating component reusability. Overloading is used in declarative languages as well as most object-oriented languages [16], and allows programmers to write polymorphic functions and methods that share the same name but respond to different parameters.

Another gain with polymorphism is the ability to treat different types as if they were of the same kind. This is common in object-oriented settings where inheritance allows subclassing of one type to another, derived type [16]. Using this kind of polymorphism allows the possibility for new, derived types to be seamlessly introduced in a system without the need to update existing components with knowledge about these new derived types. The polymorphic properties are especially powerful when abstract data types such as lists, hashes or sets are used [74].

Blobjects are type polymorphic, just like objects normally are in systems that allow subtyping [16]. Type polymorphism is useful when defining join patterns in the rule headers in the rule set in a blob, as a header can define a type and then accept any subtype to substitute that type. By allowing this, rules in a blob can be made more generic and accept specialised versions of blobjects.

3.2.4 Events and Implicit Invocation

Shaw and Garlan [64], as well as Tucker and Noonan [72], describe how communication between objects or modules in event-based systems and blackboard systems can be implicit, which means there is no need for two components to explicitly know of each other to be able to communicate. Eliminating references to other components will decrease external coupling which, as described by Eder et. al. [28], will lead to higher reusability.

According to Alghamdi [1], Stevens et. al. [69] and Sandhu et. al. [61] ridding components from references and interconnections, replacing them with implicit communication, will increase the possibilities for creating pluggable components.

Implicit invocation is used as the only means of communication between blobjects, while event-based communication can be used to describe how blobjects are passed between blobs. Blobjects use implicit invocation as they have no way of referencing each other, which leads to greater pluggability and greater reusability [1, 69, 61, 28] of blobjects.

The event-based style of communication between blobs also facilitates pluggability and reusability of entire blob hierarchies, and thus reuse of components larger and more complex than single blobjects.

3.2.5 Contexts

The idea of context-aware objects presented by Costanza and Hirschfeld [24] with context-oriented programming promises to facilitate reusability in object-oriented settings. Their model is said to enforce high cohesion by implementing all kinds of behaviour and context-awareness within the object.

We argue that a possible risk with context-oriented programming is heavy objects bloated with too much knowledge about their domain. Though facilitating the desirable effects of high cohesion—perhaps even to an extreme sense—context-oriented programming does little to rid other, more serious reusability problems in object orientation. There is, for example, a tendency towards higher coupling, no new means of increasing pluggability and no real aid for creating classes that need no altering in order to be reused.

Contexts in the blob-oriented model are based on the ideas of Costanza and Hirschfeld [24] that objects should be context-aware. However, as discussed in Section 2.2.4, context-dependent behaviour and knowledge in the blob model is implemented in the actual context rather than the object. We argue that this produces more cohesive code as there can be any number of different contexts in which a blobject can reside. By keeping context-dependent behaviour within the context there is no need for blobjects to have knowledge about every possible context available to them, thus making it possible for a blobject to be used in any context. This allows blobjects to be reused without rewriting a single line of code as long as the context implements behaviour for them.

3.2.6 Concurrency

Concurrency in itself does not really affect the reusability aspects of programming, but there are common identifiers between the two.

Properties affecting reusability in a negative way, such as coupling and message passing [1, 28, 69, 61] also affect the efficiency of a concurrent program in a negative way [4, 3, 9, 54]. Coupling and synchronous message passing hinders asynchronous execution of parallel processes, as shared data, communication and synchronisation more or less often force processes to halt and wait for other processes to finish [63].

As the blob-oriented model does not allow external coupling through referencing, nor synchronous message passing, there is reason to believe that the strong reusability means in the blob model combined with the structuring of programs in nested hierarchies, as described in Section 4.1.3, might provide means of implicit concurrency. This, however, is not examined in this thesis.

3.3 Concluding Remarks

As argued by Alghamdi [1], Stevens et. al. [69] and Sandhu et. al. [61] reusability is affected in a positive manner by general, loosely coupled independent modules with high cohesion. In many ways, object orientation facilitates writing reusable code as long as programmers explicitly design their modules for reuse. However, as argued by Doublait [27] and Biddle et. al. [11], most programmers do not design for or even consider reuse when writing their object-oriented programs, thus reusability facilities go unused and little reusable code is produced.

We argue that a better model for producing reusable code is one that makes programmers separate the general code from the system specific, without imposing limitations on the programmer's expressiveness. Introducing implicit invocation where execution is not controlled by input, but follows a logical rule-based path is also preferable from a reusability point of view, as implicit invocation means less coupling. Also, an object-oriented approach to inheritance is desirable, as even though inheritance does in some ways break encapsulation and may lead to problems with fragile hierarchical structures [77], it provides reusability means through use of polymorphism.

In the upcoming chapter we will further present and explain the blob-oriented model and relate it to the knowledge gathered in this chapter.

4

The Blob-Oriented Model

In this chapter we present the blob-oriented model through a series of illustrative figures. The blob-oriented model is a fairly small one, really only introducing two new concepts—those of a *blob* and a *blobject*. However, the blob model differs a lot from existing programming models in program composition and dispatch, thus making it meaningful to illustrate each step of constructing the blob model with a figure. Please note that *a blob* is a special *construct in* the model—not *equivalent to* the blob-oriented model.

4.1 Describing Blob Orientation

In this section we describe the blob model and explain its properties and constructs. We make use of figures to illustrate the notions of blobs and blobjects, as well as more advanced properties such as rule propagation and construction of nested hierarchies.

4.1.1 The Blob

In its simplest form a blob is just an empty object with no state and no behaviour. An empty blob, depicted in Figure 4.1, is much like an object-oriented class with no implementation. An empty blob has a type, but nothing else. A blob interacts with its surroundings through use of an inbox and an outbox, as seen in Figure 4.2. Input is received through the inbox, and the blob sends its output through the outbox.

4.1.2 The Blobject

Any object received through the inbox will be placed inside the blob, which will act as the context for the received object, called a *blobject* (see Figure 4.3). This approach to contexts differs from the one taken by Hirschfeld et. al.[35] in that the contexts are defined outside the blobjects, not within. This in turn means that every

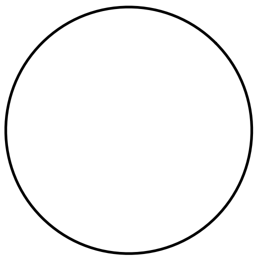


Fig. 4.1. An Empty Blob. The blob has no behaviour implementation and no content.

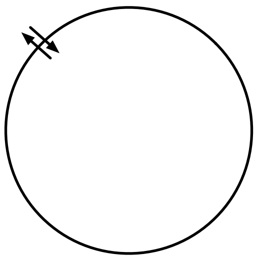


Fig. 4.2. An Inbox and an Outbox. The blob communicates through use of an inbox and an outbox.

blobject can reside in every context without having to implement behaviour for it. The behaviour of a blobject is instead defined by the context, the blob. A result of this is that a blobject of some type may behave differently depending on what blob it is currently residing in.

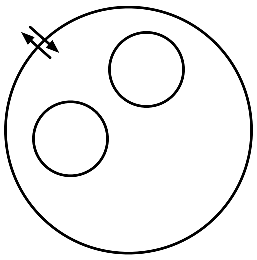


Fig. 4.3. A Blob Containing Two Blobjects. The surrounding blob act as the context for the two blobjects.

A blobject may only be contained by one blob at the time, and has no knowledge of what blob it is being contained in. Furthermore, blobjects have no knowledge of what other blobjects might be present in the blob. Blobjects kept inside a blob communicate by expelling primitive data types[1], which are then distributed to all other blobjects residing inside that blob. This may be seen in Figure 4.4. A blobject

[1] The term primitive data type is used when refering to booleans, numbers and strings.

expelling a primitive does not know what blobject might pick up that primitive, just as a blobject picking up a primitive will have no knowledge about what blobject sent it.

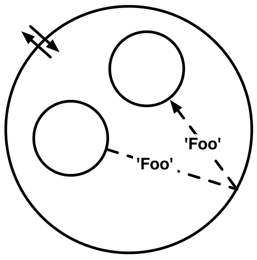


Fig. 4.4. Blobject Communication. Blobjects in a blob communicate by expelling and catching primitives.

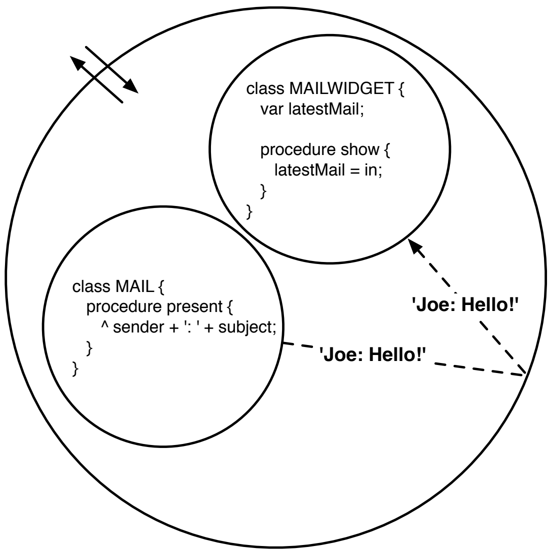
Construction of Blobjects

A blobject in a blob-oriented system is just like an object-oriented object, except that it may not handle any types more complex than primitives—strings, booleans and numbers. This reduces the external coupling described by Eder et. al. [28] by removing the possibility for interrelations through referencing between blobjects, which according to Sandhu et. al. [61] and Stevens et. al. [69] helps to raise reusability.

Blobjects may operate only on the own object and on the primitives it contains. Data slots in a blobject are readable by any part of a system, but they are only writable by the blobject itself. Manipulation of a blobject is done through use of procedures defined within the blobject, ensuring that no application can illegitimately modify the state of a blobject, as seen in Figure 4.5. This enforces encapsulation, as all manipulation of a blobject needs to be done through use of procedures. This, according to Snyder [66] and Dershem and Jipping [26] helps raise abstraction, which in turn facilitates reusability [16, 47, 58].

4.1.3 Rule Sets in Blobs

In order to control execution, a blob implements rule sets to describe how different blobject compositions should be handled. Whenever all conditions for a rule is filled, it will be executed. This is very closely related to the join patterns used in the join calculus, as described by Fournet and Gontier [29]. The blob-oriented model, however, differs from the original join calculus in that it uses implicit activation, and that join patterns may change according to contexts. In the example depicted in Figure 4.6, the implemented rule will dispatch whenever the blob contains both a `Mail` and a `MailWidget`.

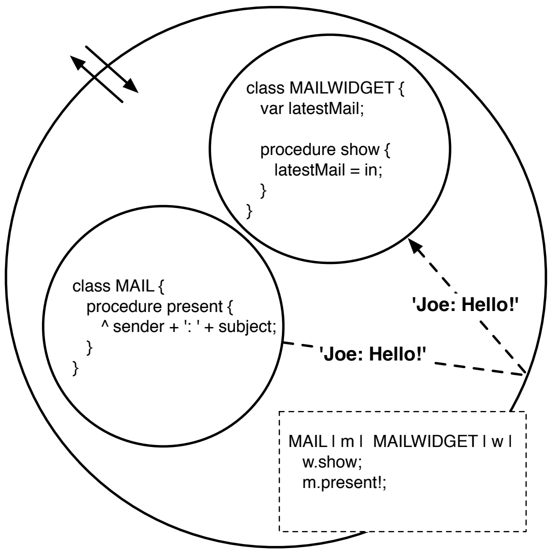


```
class MAILWIDGET {
   var latestMail;

   procedure show {
      latestMail = in;
   }
}
```

```
class MAIL {
   procedure present {
      ^ sender + ': ' + subject;
   }
}
```

Fig. 4.5. Blobject Implementation. Blobjects may contain procedures through which state may be altered.



```
class MAILWIDGET {
   var latestMail;

   procedure show {
      latestMail = in;
   }
}
```

```
class MAIL {
   procedure present {
      ^ sender + ': ' + subject;
   }
}
```

```
MAIL | m |  MAILWIDGET | w |
   w.show;
   m.present!;
```

Fig. 4.6. Rule Sets. A blob implements rule sets that dispatch whenever the conditions for the rules are filled. The rule sets are illustrated by the use of a rectangle with a dashed border. The rule set belongs to the blob which outer line it intersects.

Through the rules, a blob may directly call a procedure on a blobject matching the rule's join pattern, as is done in the second line of rule in Figure 4.6:

```
m.present!;
```

The exclamation point at the end of the call means that the procedure will be executed right there. As seen, the line of code directly above `m.present!;` does not have an exclamation point behind it:

```
w.show;
```

This makes a difference semantically, as a procedure call with no exclamation point behind it is not a direct call, but rather a call giving priority to that procedure (see Figure 4.7). The implication of this is that the next time the blobject receives a primitive, it will execute the prioritised procedure using the received primitive as a parameter, resulting in implicit invocation. As described by Alghamdi [1], Stevens et. al. [69] and Sandhu et. al. [61], this form of interaction will lead to less external coupling, and at the same time increase reusability and pluggability of components.

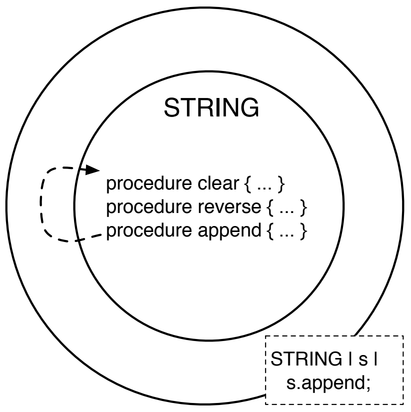


Fig. 4.7. Prioritising Procedures. Here the `append` procedure is given priority, and will be invoked the next time a primitive is distributed to the blobject.

A blob may receive blobjects through its inbox. Whenever this happens, the blob will evaluate its rule set to see if any of the implemented rules are suitable for the current situation. If a blob has two implementations matching the situation, both will be executed in the order that they are declared in the code. If one rule is a subset of the other, the super set rule will be executed alone. When a blobject is received, it is placed inside the blob. The blob rules then decide how to treat the incoming blobject. One way of handling an incoming blobject is by passing it on to one of the other blobjects kept in the blob, as is seen in Figure 4.8.

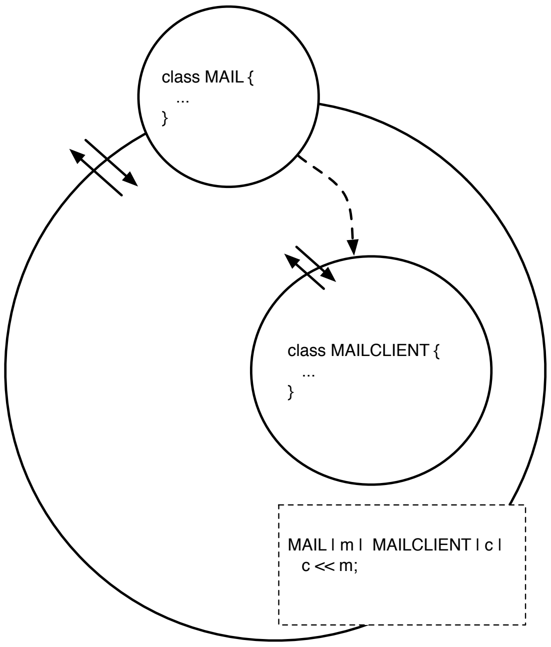


Fig. 4.8. Receiving Blobjects. When a blobject is received by a blob, it can implement rules to pass that blobject forward to another blobject it contains.

In order for the blobject to be able to receive the incoming blobject, remembering how a blobject may only handle primitive data types, the blobject must contain the incoming blobject (see Figure 4.9). This means that a blobject is a blob. This essentially means that a blob and a blobject are the same entity, but are used to represent different sides of the model. The blob corresponds to the *system specific* part of the model, while the blobject contains the *generic* parts. This separation of code coincides with Weck and Szyperski's [77] opinion that all code that can be reused should be placed in an object of its own to facilitate more code reuse.

4.1.4 Hierarchies of Nested Blobs

A blob hierarchy is constructed in the same way Knuth [42] describes a nested set, which is "*[...] a collection of sets in which any pair is either disjoint or one contains the other*" (see Figure 4.10).

As a blob may contain other blobs, that in turn may contain blobs, and so on, a blob-oriented system is formed in a hierarchical structure, where every level of the hierarchy is affected by the levels above it. If a blob at some level in the hierarchy

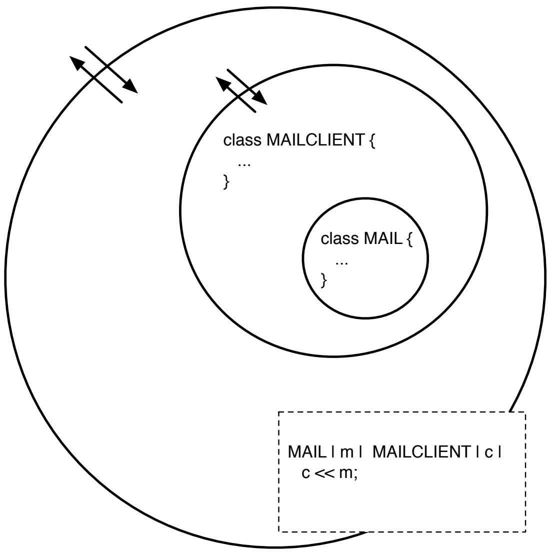


Fig. 4.9. The Blobject is a Blob. A blobject, being a blob, may contain other blobjects.

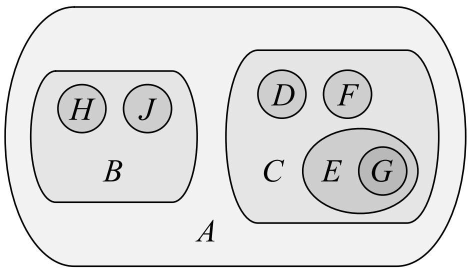


Fig. 4.10. Nested Sets. A hierarchy of nested sets as depicted by Knuth [42].

does not have an implementation of a rule to suit a specific situation, a rule from a higher level in the hierarchy is used instead. This means that rules propagate downwards, and can be ”dynamically inherited”, meaning that a blob can implement many different rule sets as it moves around in a system during program execution. The rule propagation solves the problem of extensibility in strictly layered systems described by Szyperski [71], as each new layer in a blob-oriented system may both extend and specify behaviour of higher level layers. Further, as rules propagate a layer need not implement behaviour declared in a higher level layer resulting in little code duplication, which Budd [16] and Dershem and Jipping [26] agree have a positive impact on reusability. Figure 4.11 shows a setting where the rule for how a blobject of the type `Mail` behaves in the blob `MailClient` is propagated downwards to the blob `GUI`. If the blob `GUI` was to reside outside the blob `MailClient`, it could not inherit the rule from that blob.

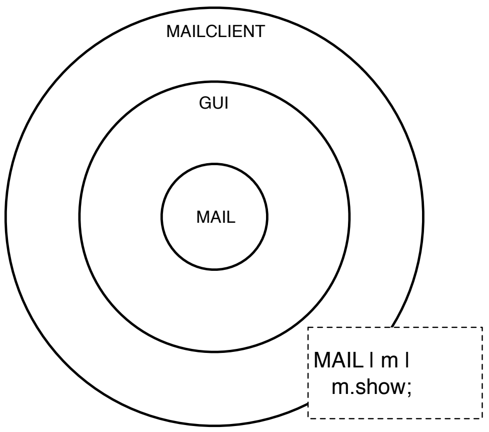


Fig. 4.11. Rule Propagation. If a blob does not have an implementation suitable for a specific situation, rules may be ”dynamically inherited” from a higher level blob.

In a blob-oriented system, any blobject that is thrown out of a blob will be sent all the way to the top of the hierarchy, where it will be garbage collected. In order to stop blobjects from being garbage collected, there are special rules called *catch rules*. The catch rule demonstrated in Figure 4.12 shows a higher-level blob catching a mail blobject that is thrown out of the lower-level blob. Catch rules differ from normal rules in that they do not propagate, as propagation of such rules could in some circumstances mean that no blobject would ever be garbage collected. The catch rule is useful when some operation needs to be performed on a blobject, i.e. sorting, and the blobject is sent into a sorting blob that performs the sort and expels the sorted blobject when it is done.

4.1.5 Aliasing and Ownership

As described by Wrigstad [80], aliasing is the ability for an object to be shared by allowing more than one concurrent owner, or reference, to it. In the blob-oriented model this is not allowed due to the strictly nested hierarchies that make up a system. Aliasing is not possible within blobjects as they have no external referencing. Aliasing is not possible within blobs either, as a blobject can only exist in one part of the system at any given time. Not only can it only exist in one part of the system, but the owner of the blobject is always the blob in which it is currently residing, thus making it impossible to have multiple owners in the system. If a blob holding a reference to a blobject passes that blobject to some other blob, the reference held by the first blob will be nullified, preserving the unique ownership.

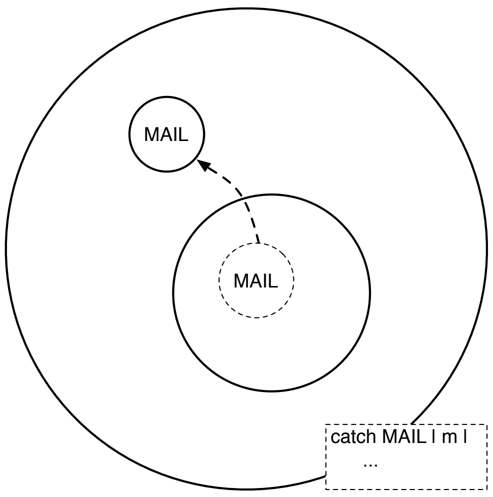


Fig. 4.12. Catch Rules. Blobjects thrown from a lower level blob may be caught and used by higher level blobs. The catch rules do not propagate like normal rules, but remain on the level that they are declared.

4.2 Concluding Remarks

The blob-oriented model is fairly small and introduces the concepts blob and blobject. The blob and the blobject are really the same entity, but are used to represent two different aspects of the model, the generic and the system specific part. Interaction in the blob-oriented model is managed through implicit invocation, and execution is controlled by rule sets defined in a blob. Rules propagate downwards, which means that a blob that does not implement a rule for a specific situation may dynamically inherit a rule from a blob higher up in the system hierarchy.

In the upcoming chapter we discuss and evaluate reusability properties in different paradigms, including the blob-oriented model. By examining examples concerning data structures, algorithms and component composition we illustrate how reusability constructs in the blob model relates to and differs from constructs in other models.

5

Evaluation

In this chapter we use examples of data structures, algorithms and component composition to compare and evaluate levels of reusability in the blob-oriented model and other software models using the measurements described in Chapter 3.

5.1 Collections

According to Weiss [79], collections are one of the most commonly reused data structures in modern programming. Therefore it is of interest to examine and compare blob-oriented collections to collections in other paradigms. The declarative approach is examined mainly because of the vital importance list processing has in functional programming [45]. When talking about lists in this chapter, we implicitly refer to the concept of *single linked lists*.

5.1.1 Declarative Collections

In most declarative languages, lists are implemented as pairs of pointers (seen in Figure 5.1). The first one—called the *head*—references a value, the second one—the *tail*—points to the next pair in the list [22]. Trees in declarative languages, depicted in Figure 5.2, are implemented using the same constructs as lists. Lists in declarative languages are built-in structures [6, 15]. According to Wadler [75] and Kühne [45], these built-in lists encapsulate the functionality of a library list structure in non-declarative languages, thus facilitating reuse in both making programs shorter but also by eliminating the risk of changes to external libraries. Furthermore, Wadler [75] says that built-in data structures eliminate the risk of unwanted side-effects, something that is also beneficial for reuse.

5.1.2 Imperative and Object-Oriented Collections

Unlike the declarative lists described in Section 5.1.1, the imperative and object-oriented lists are not built into the language, but are instead supplied to the pro-

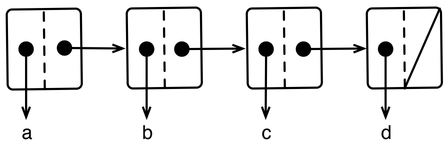


Fig. 5.1. A Declarative List. A list in a declarative language is represented as a pair of pointers, the first pointing to a value, the second pointing to the rest of the list.

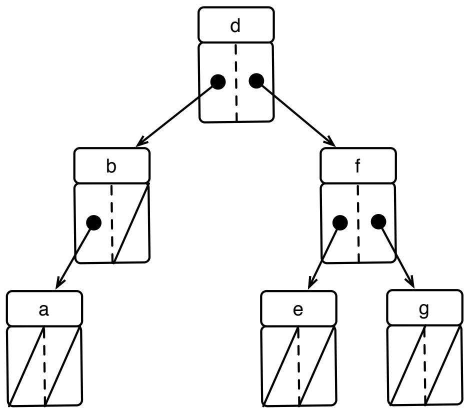


Fig. 5.2. A Declarative Binary Tree. The binary tree is constructed in a similar way to the list seen in Figure 5.1, with the use of pointers to a value and to trailing elements.

grammer as reusable components in libraries [6, 15]. This is also true for other collection data structures, such as trees (seen in Figure 5.4). Library collections and data structures, the most common examples of component reuse, are generic and can be reused over and over again in any setting that requires a specific collection or data structure [79].

Lists in imperative languages are implemented as structures containing a value and a pointer to the next element in the list as shown in Figure 5.3. They are conceptually identical to declarative lists, but instead of using language constructs, lists are constructed using reusable components. As Wadler [75] claims, this way of constructing lists is potentially harmful in comparison to built-in lists because of risks concerning potential side-effects and changes in library components.

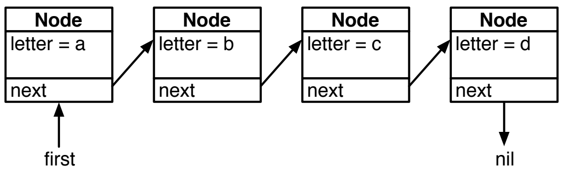


Fig. 5.3. An Imperative List. The list consists of structures that contain a value and a pointer to the next element in the list.

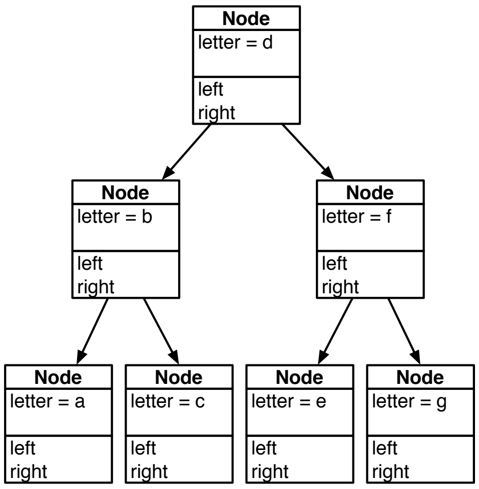


Fig. 5.4. An Imperative Binary Tree. The tree consists of node structures that contain a value and pointers to the trailing elements.

5.1.3 Blob-Oriented Collections

Blob-oriented collections differ from both the declarative collections and the imperative ones in that there is no explicit construct for collections in the blob model. Instead, as depicted in Figure 5.5 and Figure 5.6, lists and trees are created using the same constructs used to form nested hierarchies of blobs. Lists and trees are in fact nested hierarchies—just like any other component in the model. Knuth [42] notes that any hierarchical structure forms a list, which makes the notion of lists and trees as data structures in the blob-oriented model more a matter of semantic perception in general than an explicit construct. In some ways though, the blob-oriented solution is very much like the declarative one, as each element in the list has a value and a pointer to the rest of the list.

Additional behaviour is added to a list or a tree structure by placing it inside a blob that implements rules describing the desired behaviour. A list is sorted by placing an unsorted list in a sorting blob that implements rules for sorting.

As no external structure is needed to form collections, the blob-oriented model enjoys the reusability properties of declarative languages described by Wadler [75]. Further, the in-built representation of lists makes the blob model less dependent on backward compatibility, which Wadler [75] also describes as favoring reuse.

5.1.4 Comparing Notes

The blob-oriented model does not have a special construct for collection representation, and this favors reusability when it comes to backwards compatibility and

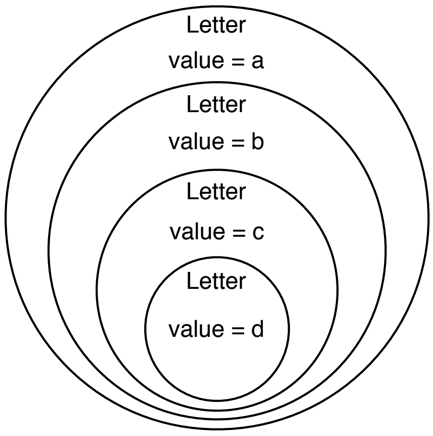


Fig. 5.5. A Blob-Oriented List. As everything in the blob model is constructed as nested hierarchies, a list is easily created without the use of any special list constructs.

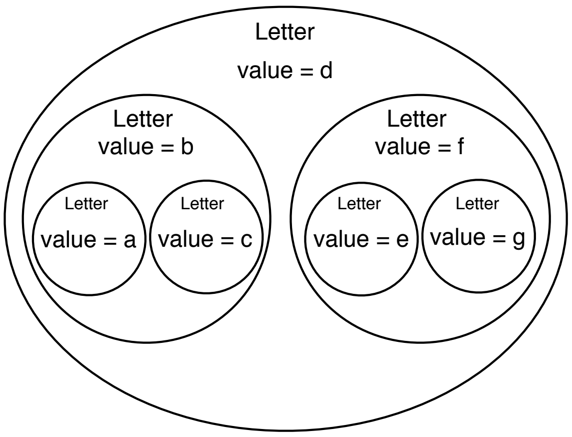


Fig. 5.6. A Blob-Oriented Binary Tree. Using nested sets, a tree of letters can be created without any special constructs for a tree data type.

correctness of special language constructs or library components [75]. Yet, the blob model lacks the abstraction provided by both the imperative and the declarative solutions, as there is little separation between a single blobject and a list of them. However, the possibility of enclosing a blob list in a blob implementing list behaviour in some ways raises abstraction, as the list then may be viewed as a module rather than just a hierarchical structure.

As behaviour of a collection in a blob-oriented system is placed in a blob (or several blobs) surrounding the collection, the blob-oriented collections are somewhat less cohesive than both the imperative and the declarative collection structures, where behaviour is directly attached to the construct itself [6, 15]. Yet, this solution makes the blob-oriented collections more flexible than the declarative and imperative ones, as behaviour can be easily attached or changed whenever needed during program execution. Furthermore, and most importantly, declaring list behaviour in separate blobs means that any list in any blob-oriented system will adapt the declared behaviour when placed inside that blob, without the need for extending behaviour through inheritance or by use of macros. This makes blob-oriented collections less externally coupled than the declarative or imperative collections, while at the same time better facilitating reuse for specific collection behaviour.

5.2 Key Word In Context

The *Key Word in Context* problem (KWIC) was described by Parnas [57] in his 1972 article ”On the Criteria To Be Used in Decomposing Systems into Modules”:

> “The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be ’circularly shifted’ by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.”

Even though fairly small, the KWIC problem is an interesting one as it highlights the problems of tool abstraction without it being overwhelming in size [32]. Parnas [57] originally used the problem to analyse how different solutions to the problem could withstand changes to the processing algorithm or to the data structures. Later, Garlan et. al. [32] extended the analysis by also considering how a solution could enhance the system with added functionality, performance levels and the extent to which components in the system could be considered reusable. A module was considered reusable if it could be used in other settings without any major alterations to it. The line storage, for example, was considered reusable if it could be used to store other lists of words in another system [32].

We present three different solutions to the KWIC problem, one object-oriented, one using implicit invocation and one blob-oriented and examine each solution from a reusability perspective.

5.2.1 An Object-oriented Solution

An object-oriented solution to the KWIC problem described by Shaw and Garlan [64], shown in Figure 5.7, proposes the use of interfaces for communication between modules. The solution consists of functionality for handling i/o, shifting and alphabetising.

By encapsulating functionality in modules, changes may be made to individual modules without affecting others, as long as their interface is kept intact. This solution is, according to Shaw and Garlan [64], fit to handle design changes as the way data is represented in a module is transparent to other modules. The same concept applies for algorithms, as the only abstraction of a module that another module needs to have knowledge about is the interface. Thus, changes to data representation and algorithms in a module can be made without affecting the rest of the system.

Though a well suited model for coping with design changes, Garlan et. al. [32] identifies this approach to the KWIC problem as being poor when it comes to handling certain functional changes. They claim that problems could arise when adding new behaviour to the system, as existing modules need to be either updated with new functions or implement knowledge about modules containing new functionality.

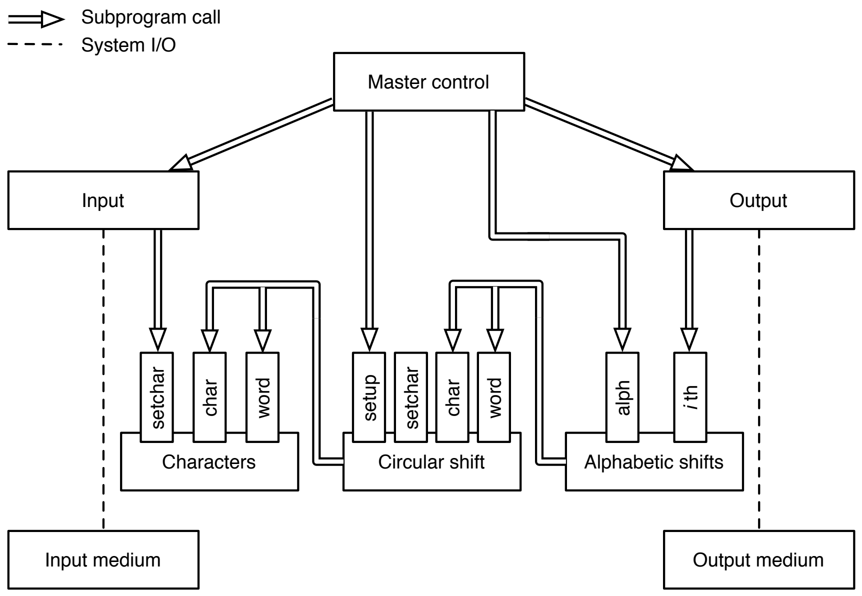


Fig. 5.7. KWIC: Object-Oriented Solution. An object-oriented solution to the KWIC problem as proposed by Shaw and Garlan [64].

5.2.2 Implicit Invocation

A solution using implicit invocation, described by Shaw and Garlan [64], is depicted in Figure 5.8. The implementation works by implicitly invoking computations whenever data in the system changes. A circular shift is implicitly triggered by addition of a new line, which in turn causes an event to be sent to the alphabetiser that will respond to the event with an alphabetic shift.

This solution facilitates reuse in that modules need only be reliant on events, instead of other modules [64]. The circular shifter and the alphabetiser could be seamlessly removed or exchanged without any alterations to other system modules. Shaw and Garlan [64] note that the use of implicit invocation might make it more difficult to control the execution flow in such a system.

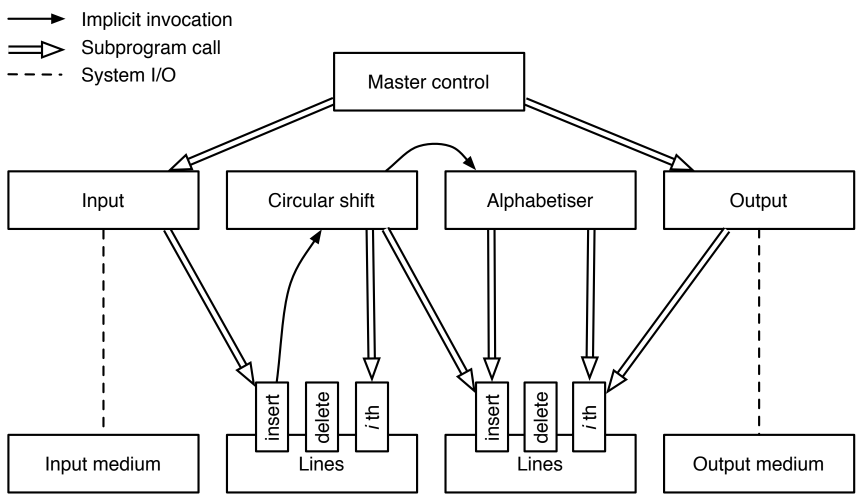


Fig. 5.8. KWIC: Implicit Invocation Solution. An implicit invocation solution to the KWIC problem as proposed by Shaw and Garlan [64].

5.2.3 A Blob-Oriented Solution

A blob-oriented solution to the KWIC problem is depicted in Figure 5.9. The solution works by placing a list of lines at the bottom of a blob hierarchy. The modules will work from the inside out, gradually closing in on the solution. The shifter works by creating permutations of all possible line shifts, storing the results in result blobjects that inherit from the blobject class `Line`. The result blobjects are passed upwards where they are caught by the alphabetiser, which places them in a sorted hierarchical list before passing them onward. The list is caught by the printer, which will perform the system output task on the list of result blobjects. Once the printer is done, it will pass the result blobjects outwards, where the main blob will catch them and end execution.

In the blob-oriented solution, no module is dependent or even aware of another[1], as they all work by simply catching result blobjects coming from a level below. It would be entirely possible to remove any blob without making any alterations to the rest of the system. For example, removal of the alphabetiser would result in all permutations of all lines being printed in the order they were generated. Because the

[1] This is not true for the main blob, as it is responsible for creating the hierarchy and thus need to be aware of what blobs to create.

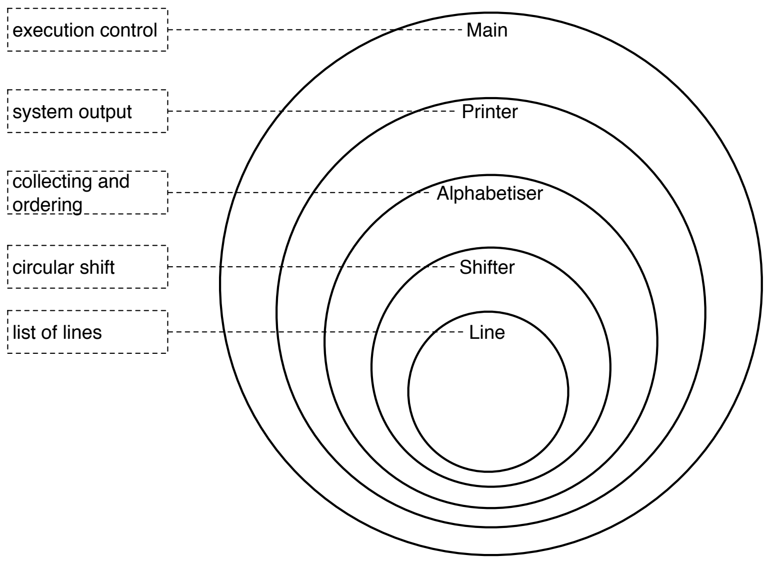


Fig. 5.9. KWIC: Blob-Oriented Solution. The blob-oriented solution uses layers of responsibility to solve the problem.

result blobject is a subclass of the line blobject, removing the shifter would result in all lines being alphabetised and printed, even if no result blobjects were generated.

5.2.4 Comparing Notes

In the KWIC problem domain the solutions illustrated in this chapter using object-orientation and implicit invocation are modularised, thus offering abstraction on both design level and implementation level. The proposed blob-oriented solution is also modularised and offers the same kind of abstraction. However, as the blob-oriented solution consists of hierarchically nested components, abstraction can be raised and lowered by choosing what layer to examine. In such systems, each layer provides increased levels of abstraction, and thus greater means of understanding the roles of each individual layer [71], making the blob-oriented solution a bit more powerful than the others when it comes to abstraction.

When examining cohesion in the three solutions it is clear that the modularised object-oriented solution as well as the modularised implicit invocation-based solution are more cohesive than the blob-oriented solution, at least at first glance. As the non-blob based solutions provide both data structures and behaviour explicitly within the module, they can be considered highly cohesive in comparison. Behaviour in the blob model is, as described in Section 4.1.3, always specified in a context and thus not within the blob or blobject for which the behaviour applies. This is true also for the blob-oriented KWIC solution, and thus it is less cohesive. However, as stated in Section 3.2.5, we claim that by keeping behaviour separated from data structures, and instead implementing it in a certain context, overall cohesion is raised. By allowing different behaviour in different contexts, it would be much less cohesive to implement context-dependent behaviour anywhere else than in the actual context.

Examining degrees of coupling in the proposed solutions, the object-oriented one is by far the most coupled. Both the implicit invocation-solution as well as the blob-oriented solution use implicit—non-coupled—communication, thus reducing the need for them to implement knowledge about other modules' interfaces. In contrast, Garlan et. al. [32] describe how the explicit communication in the object-oriented solution is likely to cause problems when extending the object-oriented model with new behaviour. As modules either need to be updated with new behaviour or with knowledge about new modules that implement the new behaviour, we come across the age-old object-oriented problem of balancing between low coupling and high cohesion. In the implicit invocation-solution and the blob-oriented solution, however, components are easily pluggable without affecting the system, thus allowing modules to be more easily reused. Comparing the two latter solutions, the blob-oriented one provides even greater means of pluggability, as all modules make use of implicit invocation and are pluggable on basically the same premises. The blob-oriented model allows the output module, for example, to be arbitrarily plugged in or out, while the implicit invocation-solution does not.

5.3 Components

Szyperski [71] claims independent components facilitate practical reuse of software, which makes component composition an interesting method of evaluating reusability in software systems. According to Szyperski [71], with support from Doublait [27], software needs modularity in architecture and design as well as implementation, thus realising the need to view reusability on a higher level than data structures and algorithms.

In this section we examine and evaluate a higher-level architectural design of an e-mail client in order to grasp reusability aspects in composition and pluggability of components. The e-mail client communicates with a server from where it receives and sends e-mail. A virus checker and a spam filter are applied to the e-mail before it is presented in a graphical user interface.

5.3.1 Object-Oriented Components

The object-oriented e-mail client design we propose consists of a `Controller` class that handles execution flow. The controller communicates with the `ServerChecker` in order to send and receive e-mail. The controller also communicates with the `GUI` in order to present incoming e-mail to the user and send e-mail that the user composes using the graphical user interface.

A spam filter is plugged in to the controller, which lets the controller check all incoming e-mail for spam and sort or flag the mail as spam before presenting it to the user.

The `VirusChecker` is installed using the *adapter* pattern [50] between the controller and the server checker, so that all incoming and outgoing e-mail traffic is scanned for viruses. This setup allows the controller and the server checker to stay oblivious of the virus checker, which makes it seamlessly removable, extendable and exchangeable.

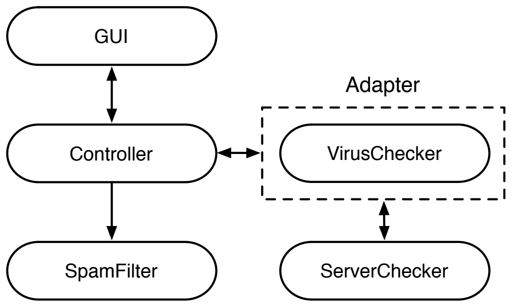


Fig. 5.10. An Object-Oriented E-mail Client.

5.3.2 Strictly Layered Components

We suggest an implementation of a strictly layered e-mail client system, shown in Figure 5.11. The bottom most layer—the server checker—is the one closest to the hardware, and the top most layer—the graphical user interface—handles the communication with the user. Each layer is directly dependent upon at least one other layer, which means that no layer could be seamlessly removed without having to make alterations to some of the other layers. Also, introducing a new layer between the spam filter and the virus checker would entail making alterations to both these layers.

5.3.3 Blob-Oriented Components

The blob-oriented design of the e-mail client, depicted in Figure 5.12, is much like the layered solution described in Section 5.3.2. The two differs in that the blob-oriented components have no interrelations, making them seamlessly interchangeable and removable. Removing the virus checker, for example, would pose

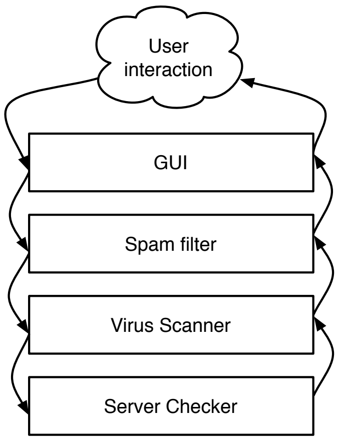


Fig. 5.11. A Strictly Layered E-mail Client.

no problem—the e-mail would simply be passed directly from the server checker to the spam filter. Also, the blob-oriented solution does not suffer from the problem of extensibility that, as discussed by Szyperski [71], is found in strictly layered systems.

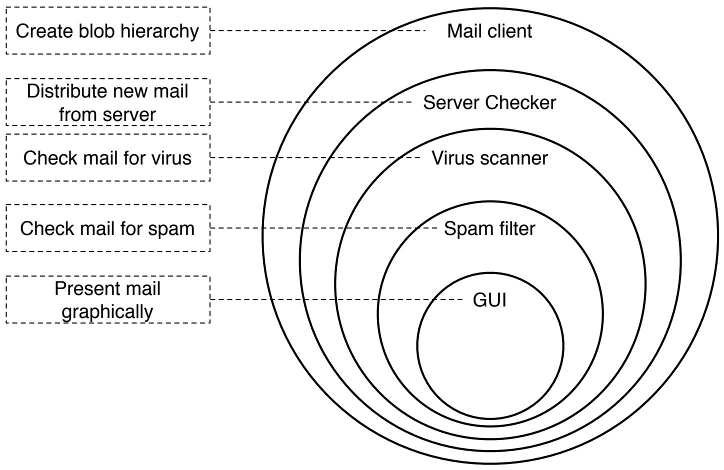


Fig. 5.12. A Blob-Oriented E-Mail Client.

All components in the blob-oriented system are kept inside a mail client blob, responsible for setting up the blob hierarchy needed for correct execution. Also, the mail client blob implements rules for distribution of mail that will propagate downwards to all blobs it contains. As a result, the virus scanner for example, will not need to have any knowledge about the spam filter in order to send mail to it, as the rule for this behaviour is propagated down from the mail client blob where it is implemented. Any mail received by the server checker will travel down the hierarchy until it reaches the graphical user interface, where it will be presented to the user. Whenever an e-mail is sent, it will travel up the hierarchy until it is caught by the server checker and sent to the server.

5.3.4 Comparing Notes

In both the blob-oriented and the layered solution the explicit layering of the designs shows that an e-mail must pass the spam filter and the virus checker before reaching the graphical user interface. This makes these two solutions somewhat superior to the object-oriented solution when it comes to abstraction. However, this could be remedied by applying the adapter pattern [50] to the spam filter in the object-oriented solution in the same way as is done to the virus checker, yielding a more chain-like design.

One of the major flaws in the blob-oriented solution is the cohesion problem that arises when the top most blob implements rules for e-mail distribution, propagating them to the lower level blobs. However, this eliminates code duplication as distribution code need only be written once, instead of four times. The object-oriented system also suffers from cohesion problems, especially in the controller class that is likely to often act simply as a router for incoming and outgoing mail.

The major difference between the three suggested systems is the amount of coupling found in them. Even though built to support loosely coupled components [8, 64], the layered implementation performs the worst, as both the spam filter and the virus checker are externally coupled to two other components. The layered system is also very inflexible, changing the order of the virus checker and the spam filter would entail changes to all components in the system. The object-oriented solution is not much better when it comes to coupling, only the virus checker is truly removable, and changing the order of the spam checking and the virus control would entail redesigning the entire system. The blob-oriented model, however, performs very well from this perspective. As no module other than the main one has knowledge of any of the other modules, all blobs are seamlessly interchangeable, movable and removable. Introducing new blobs in the system would pose no problem as no blob is dependent upon another. Switching the order of virus checking and spam filtering would simply be a matter of placing them in a different order when constructing the hierarchy—no other changes would need to be made, and the system would still function.

5.4 Evaluation of Reuse in the Blob-Oriented Model

As can be seen in all three examples above, the blob-oriented model performs at least as well as other models do when it comes to abstraction. The blob model is highly scalable as all levels in a blob hierarchy represent a new level of abstraction. The e-mail client implementation in Section 5.3.3 shows how an e-mail must pass through both a virus checker and a spam filter in order to reach the graphical user interface, something which illustrates that the order of the levels also brings abstraction to the model by defining both internal and external domain boundaries. The lack of an explicit data structure that represents collections does in some ways lower abstraction, but as the built-in collection representation—nesting of blobs—favors reuse when it comes to backwards compatibility [75], we argue that this is not a great reuse problem. Also, this problem may be remedied by placing the collection hierarchy in a collection blob wrapper and thereby raising abstraction.

One of the biggest issues when it comes to reusability in the blob-oriented model is that of potentially low cohesion in blob rules. As rules propagate, one may be tempted to simply place all rules in a main blob surrounding all others, moving all system specific code to one blob. Doing this would rid the blob-oriented model of its context dependent properties, which very likely would be undesirable. One way of dealing with this issue would be to remove the rule propagation, which would ensure that all rules belonging to a context was declared in that context. On the other hand, rule propagation minimises code duplication, which according to both Budd [16] and Dershem and Jipping [26] favors reuse. Therefore, we argue that rule propagation is still a desirable feature in the blob-oriented model.

Finally, as seen specifically in the KWIC solution and the e-mail client implementation, the blob-oriented model facilitates writing loosely coupled and highly flexible software components, something which Alghamdi [1], Stevens et. al. [69] and Sandhu et. al. [61] all agree favors both reuse and software quality. The loose coupling between components is partially made possible by the rule propagation discussed earlier, as all rules that demand components to know about one another may be declared in a higher level blob.

5.5 Concluding Remarks

We have shown three different implementations of three different problems, and have compared them to each other using the measurements defined in Chapter 3. We have concluded that the blob-oriented model performs at least equally well compared to other models when it comes to abstraction, but that it suffers from a slight cohesion problem. Finally, we have explained how the facilities for producing loosely coupled components in the blob-oriented model raises code and component reusability.

In the upcoming chapter we present critique of the blob-oriented model, outline future research directions and conclude our thesis.

6 Conclusions

In this final chapter we conclude our work and present critique of the blob-oriented model. Finally, we outline future research directions in the continued evolution of the blob-oriented model.

6.1 Criticism

In this section we present and discuss critique on the blob-oriented model. This is done primarily to help future research by highlighting problems in the blob model that still remain unresolved.

6.1.1 Absence of Aliasing

Avoiding aliasing and enforcing unique ownership of blobjects, as discussed in Section 4.1.5, will rid problems created by aliasing, for example ripple effects [36, 80]. However, avoiding aliasing will also create problems. Not being able to alias blobjects will make it more complex to handle commonly used data structures [80]. Keeping blobjects sorted differently in two separate lists, for example, is impossible without either copying blobjects or storing their hash id in the lists instead of the actual blobject.

6.1.2 Implicit Invocation

Although favoring reuse, implicit invocation is not without disadvantages [64]. One of the main problems with implicit invocation is added difficulty in execution tracking and control [64]. In addition to this, even if not true for all implementations of implicit invocation systems, implicit invocation can be the cause of increased overhead yielding performance issues [13, 64].

6.1.3 Cohesion Issues

Though the blob model is said to enforce reusability and very much offers great means for creating reusable code, there is a cohesion problem with the system specific rule sets. As the rules propagate, it would be comfortable to just place most or all rules at the top of a hierarchy and let them trickle down through the system. This, however, would mean that sub-hierarchies could not be reused as modules. Some amount of incohesieveness in favor of propagating rules is still advisable though, as rule propagation means less code duplication which in turn has a positive impact on reusability [16, 26].

6.1.4 Reusability Issues

Though we conclude that the blob-oriented model does favor reusability to a greater extent than the other paradigms and models we discuss in this thesis, we have yet to overcome a few issues. For example, we claim that the real strength of the blob model when it comes to reuse is the ability to reuse source code and modules. Though this might be true, production of source code and modules are only two of many processes in software development where reusability can be applied [27]. We have yet to describe how the blob-oriented model handles reuse in requirements specification, system analysis, testing, documentation and other steps in production.

Also, because the blob-oriented model does have a problem with cohesion in rule sets, problems with scattered or seemingly misplaced code could occur, thus confusing programmers looking to reuse source code.

Finally, by keeping the object-oriented inheritance the blob model is vulnuarable to the problems with inhertitance identified by Weck and Szyperski [77]. However, we argue that the reusability profits gained through not excluding inheritance from the model, such as the polymorphic properties and minimising of code duplication, outweighs the negative aspects of inheritance from a reusability point of view.

6.2 Summary of Conclusions

We have presented the blob-oriented model as a means of producing reusable software without having to explicitly plan for reuse. The blob model separates generic behaviour from system specific behaviour, and uses implicit invocation to control blobject interaction. The blob-oriented model also introduces rule sets that describe context-dependent behaviour.

In order to evaluate our work we have defined three software reusability metrics—coupling, cohesion and abstraction. We have examined existing software paradigms and architectures to isolate properties that strengthen reuse, and discussed what effect these properties had on our metrics in a software system. We have evaluated the reusability aspects of the blob model by comparing solutions to three software problems using the three metrics, and found that the blob-oriented model performs very well when it comes to increasing reusability by decreasing external coupling.

Finally, we have presented critique of the model and in the upcoming section we outline future research directions.

6.3 Future Work

In this section we suggest a few different approaches towards further investigating the blob-oriented model. We also present a few thoughts on how to tackle these problems.

6.3.1 Implementation

A natural step following this thesis is to make an implementation of the blob programming model. We believe that the blob-oriented model would benefit from having an object or code browser, much like the Smalltalk desktop user interface [33]. Because the blob model forces the programmer to separate generic and specific code, an object will need two different sets of code, something that could be confusing or even unmanageable without a tailored IDE-like tool.

We make no assumptions to whether or not an implementation is statically or dynamically typed, or to any other religious properties that might need consideration, as we realise that such guidance from our side is not really relevant.

6.3.2 Implicit Concurrency

In this thesis we have brushed upon the prospect of having means of implicit concurrency in the blob-oriented model. We strongly encourage a deeper examination of this possibility, as we believe there are a number of factors in the blob model that could enable such a feature. For one, the forced structuring of programs in nested hierarchies where forks are not allowed to communicate is a perfect example of an environment where concurrency could thrive. Second, the use of implicit invocation allows processes to execute without having to wait for a receiver to acknowledge that a message has been received. This relieves the system from synchronous communication, which could be highly favorable for incorporating implicit concurrency into the blob model. Third, the use of join patterns are already present and could provide a base for further expanding the means of concurrency in the blob-oriented model.

6.3.3 Formalisation

Though most parts of the new blob-oriented model presented in this thesis are explained, defended and exemplified, there are still room for improvements. By formalising the blob-oriented model, its consistency and validity can be checked to a much greater extent than we have been able to in this thesis.

References

1. J. Alghamdi. Measuring Software Coupling. 2007. Proceedings of the 6th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, Corfu Island, Greece.
2. G. Andrews and F. Schneider. Concepts and Notations for Concurrent Programming. *Computing Surveys, vol. 15, no. 1*, 1983.
3. J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, 2003.
4. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG, 2nd edition*. Ericsson, Telecommunications Systems Laboratories, 1996.
5. J. Backus. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM, vol. 21, no. 8*, 1978.
6. H. Bal and D. Grune. *Programming Language Essentials*. Addison-Wesley, 1994.
7. E. Baniassad and S. Fleisner. The Geography of Programming. 2006. OOPSLA '06 Proceedings.
8. D. Batory and S. O'Malley. The Design and Implementation of Hierarchial Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology, vol. 1, no. 4*, 1992.
9. M. Ben-Ari. *Pinciples of Concurrent and Distributed Programming*. Prentice Hall, 1990.
10. N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for $C^{\#}$. *ACM Transactions on Programming Languages and Systems, vol. 26, no. 5.*, 2004.
11. R. Biddle, E. Tempero, and P. Andreae. Object-Oriented Programming and Reusability. Technical report, Victoria University, Wellington, Department of Computer Science, 1995.
12. C. Boldyreff. Reuse, Software Concepts, Descriptive Methods and the Practitioner Project. *Software Engineering Notes, vol. 14, no. 2*, 1989.
13. J. Bosch. *Design and Use of Software Architecture*. Pearson Education, Ltd., 2000.
14. L. Briand, J. Daly, V. Porter, and J. Wust. A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems. Technical report, Fraunhofer Institute for Experimental Software Engineering. Kaiserslautern, Germany, 1998.
15. T. Budd. *Classic Data Structures in Java*. Addison-Wesley, 2001.
16. T. Budd. *An Introduction To Object-Oriented Programming, 3rd edition*. Addison-Wesley, 2002.

17. L. Capretz. A Brief History of the Object-Oriented Approach. *ACM Sigsoft, Software Engineering Notes vol. 28, no. 2*, 2003.
18. D. Card, V. Church, and W. Agresti. An Empirical Study of Software Design Practices. *IEEE Transactions on Software Engineering, vol. 12, no. 2*, 1986.
19. D. Card and F. McGarry. The Software Engineering Laboratory. Technical report, NASA, 1982.
20. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys, vol. 17, no. 4*, 1985.
21. K. Clark and S. Tärnlund. *Logic Programming*. Academic Press, Inc., 1982.
22. R. Clark and L. Wilson. *Comparative Programming Languages, 3rd edition*. Addison-Wesley, 2001.
23. D. Corkill. Blackboard Systems. *AI Expert 6, no. 9*, 1991.
24. P. Costanza and R. Hirschfeld. Language Constructs for Context-Oriented Programming. 2005. ACM Dynamic Languages Symposium 2005, Proceedings, ACM Press.
25. P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming. 2005. ACM Dynamic Languages Symposium 2005, Proceedings, ACM Press.
26. H. Dershem and M. Jipping. *Programming Languages Structures and Models, 2nd edition*. PWS Publishing Co., 1995.
27. S. Doublait. Standard Resuse Practices: Many Myths vs. a Reality. *StandardView, vol. 5, no. 2*, 1997.
28. J. Eder, G. Kappel, and M. Schrefl. Coupling and Cohesion in Object-Oriented Systems. Institut für Informatik, Universität Klagenfurt, Institut für Informatik, Universität Linz and Institut für Witschaftsinformatik, Universität Linz, 1992.
29. C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. INRIA Rocquencourt, 1995.
30. W. Frakes and C. Terry. Software Reuse: Metrics and Models. *ACM Computing Service, vol. 28, no. 2*, 1996.
31. R. Gabriel. Objects Have Failed. 2002. OOPSLA '02 Debate.
32. D. Garlan, G. Kaiser, and D. Notkin. Using Tool Abstraction To Compose Systems. *IEEE Computer, vol. 25, no. 6*, 1992.
33. M. Guzdial and E. Soloway. Teaching the Nintendo Generation to Program. *Communications of the ACM, vol. 45, no. 4*, 2002.
34. P. Hansen. Concurrent Programming Concepts. *Computing Surveys, vol. 5, no. 4*, 1973.
35. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context Oriented Programming. *Journal of Object Technology, vol. 7, no. 3*, 2008.
36. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the Treatment of Object Aliasing. *SIGPLAN OOPS Mess., vol. 3, no. 2*, 1992.
37. C. Hogger. *Introduction to Logic Programming*. Academic Press, Inc., 1984.
38. G. Itzstein and M. Jasiunas. On Implementing High Level Concurrency in Java. In *Advances in Computer Systems Architecture*. Springer Verlag, 2003.
39. R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming, vol. 1, no. 2*, 1988.
40. G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241. Springer-Verlag, 1997.

41. G. Kiczales, J. Lamping, C. Lopes, J. Hugunin, E. Hilsdale, and C. Boyapati. *Aspect-Oriented Programming*. United States Patent, number 6,467,086, 2002.
42. D. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968.
43. R. Kowalski. Logic as a Computer Language. In K. Clark, S. Tärnlund, editor, *Logic Programming*, pages 3 – 16. Academic Press, 1982.
44. G. Krasner and S. Pope. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming, vol. 1, no. 3*, 1988.
45. T. Kühne. Recipes to Reuse. 1996. Proceedings of EuroPLoP.
46. J. Leeuwen and J. Wiedermann. The Turing Machine Paradigm in Contemporary Computing. Partially supported by GA CR grant no. 201/98/0717 and by EC Contract IST-1999-14186.
47. J. Lewis, S. Henry, D. Kafura, and R. Schulman. An Empirical Study of the Object-Oriented Paradigm and Software Reuse. 1991. OOPSLA '91 Proceedings.
48. L. Mandel and L. Maranget. Programming in JoCaml — Extended Version. Technical report, INRIA Rocquencourt, 2007.
49. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.
50. R. Martin and M. Martin. *Agile Principles, Patterns and Practices in $C^{\#}$*. Prentice Hall, 2007.
51. J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1. *Communications of the ACM, vol. 3, no. 4*, 1960.
52. J. McCarthy, P. Abrahams, D. Edwards, T. Hart, and M. Levin. *Lisp 1.5 Programmer's Manual, 2nd edition*. The Computation Center and Research Laboritory of Electronics, MIT, 1985.
53. R. Milner. Elements of Interaction. *Communications of the ACM, vol. 36, no. 1*, 1993.
54. K. Nielsen. Task Coupling and Cohesion in Ada. *Ada Letters, vol. 6, no. 4*, 1986.
55. O. Papapetrou and G. Papadopoulos. Aspect-Oriented Programming for a Component-Based Real Life Application: A Case Study. *ACM Symposium on applied computing*, 2004.
56. Parallab, Bergen Center for Computational Science, and University of Bergen. Software Reusability and Efficiency. Technical report, ENACTS, 2004.
57. D. Parnas. On the Criteria to be Used in Decomposing Systems Into Modules. *Communications of the ACM, vol. 15, no. 12*, 1972.
58. W. Pree. Reusability Problems of Object-Oriented Software Building Blocks. Technical report, Institut für Wirtschaftsinformatik, supported by Siemens AG, Munich, 1991.
59. R. Sebesta. *Concepts of Programming Languages, 8th edition*. Addison-Wesley, 2008.
60. A. Rockley, P. Kostur, and S. Manning. *Managing Enterprise Content*. New Riders Publishing, 2003.
61. P. Sandhu, P. Blecharz, and H. Singh. A Taguchi Approach to Investigate Impact of Factors for Reusability of Software Components. *International Journal of Applied Science, Engineering and Technology, vol. 4, no. 1*, 2008.
62. T. Schorsch and D. Cook. Evolutionary Trends of Programming Languages. *CROSSTALK, The Journal of Defense Software Engineering*, 2003.

63. R. Sethi. *Programming Languages Concepts and Constructs, 2nd edition*. Addison-Wesley, 1997.
64. M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, Inc., 1996.
65. D. Shukla, S. Fell, and C. Sells. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. *MSDN Magazine, no. 3*, 2002.
66. A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. 1986. OOPSLA '86 Proceedings.
67. I. Sommerville. *Software Engineering, 6th Edition*. Addison-Wesley, 2001.
68. F. Steimann. The Paradoxical Success of Aspect-Oriented Programming. 2006. OOPSLA '06 Proceedings.
69. W. Stevens, G. Myers, and L. Constantine. Structured Design. *IBM Systems Journal, vol. 13, no. 2*, 1994.
70. G. Succi, C. Uhrik, and M. Ronchetti. Reusability and Portability of Logic Programming. *Journal of Programming Languages*, 1996.
71. C. Szyperski. *Component Software*. Pearson Education, Ltd., 1999.
72. A. Tucker and R. Noonan. *Programming Languages Principles and Paradigms*. The McGraw-Hill Companies, Inc., 2002.
73. J. Udell. Componentware. *BYTE vol. 5*, 1994.
74. M. Vachharajani, N. Vachharajani, and D. August. A Comparison of Reuse in Object-Oriented Programming and Structural Modeling Systems. Departments of Computer Science and Electrical Engineering, Princeton University.
75. P. Wadler. How to Solve the Reuse Problem? Functional Programming. 1998. ICSR '98: Proceedings of the 5th International Conference on Software Reuse, IEEE Computer Society.
76. D. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall Internation, Ltd., 1990.
77. W. Weck and C. Szyperski. Do We Need Inheritance? ETH Zurich & Queensland University of Technology.
78. P. Wegner. Why Interaction is More Powerful than Algorithms. *Communications of the ACM, vol. 40, no. 5*, 1997.
79. M. Weiss. *Data Structures and Problem Solving using Java, 2nd edition*. Addison-Wesley, 2002.
80. T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Stockholm, 2006.