An object model with feature-based dispatch for context-sensitive behavior in distributed systems

Eddy Truyen
Wouter Joosen

Report CW 527, October 17, 2008

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

An object model with feature-based dispatch for context-sensitive behavior in distributed systems

Eddy Truyen
Wouter Joosen

Report CW 527, October 17, 2008

Department of Computer Science, K.U.Leuven

Abstract

Feature-based composition essentially complements object-oriented composition by enabling the coherent extension of multiple entities (objects, classes) such that an integral feature can be the subject of composition – rather than fragments of features. The conventional object model of state-of-the-art OO programming languages does not incorporate the notion of feature dispatch however, and, therefore, the application developer must design and implement it by hand.

This paper presents an enhanced object model that incorporates feature-based dispatch, based on the notion of feature identifiers. Relying on this model, message senders can affect feature dispatch by specifying the required features on the basis of a simple identifier. In this context, sender-initiated late binding is particularly relevant when clients customize the functionality of remote services by selecting feature identifiers instead of manipulating remote references in complicated middleware.

An Object Model with Feature-Based Dispatch for Context-Sensitive Behavior in Distributed Systems

Eddy Truyen and Wouter Joosen
Dept. of Computer Science
K.U.Leuven. Belgium
Eddy.Truyen@cs.kuleuven.be

October 17, 2008

Abstract

Feature-based composition essentially complements object-oriented composition by enabling the coherent extension of multiple entities (objects, classes) such that an integral feature can be the subject of composition – rather than fragments of features. The conventional object model of state-of-the-art OO programming languages does not incorporate the notion of feature dispatch however, and, therefore, the application developer must design and implement it by hand.

This paper presents an enhanced object model that incorporates feature-based dispatch, based on the notion of feature identifiers. Relying on this model, message senders can affect feature dispatch by specifying the required features on the basis of a simple identifier. In this context, sender-initiated late binding is particularly relevant when clients customize the functionality of remote services by selecting feature identifiers instead of manipulating remote references in complicated middleware.

1 Introduction

The state-of-the-art in software development for large and complex applications has evolved towards supporting increasingly complex distributed software systems, running on heterogeneous platforms, meeting various non functional requirements ranging from high-availability and security towards modifiability and user-friendliness. This evolution has clearly challenged our overall vision on software systems and the way these systems should be developed.

Traditional software engineering methodologies have been focusing on initially creating a system that meets functional requirements through OOA and

OOD activities, assuming that complex non-functional requirements can and should be treated in a second phase. More recently, software development is considered to be a process in which competing requirements must be met by designing separate and modular solutions for various categories of requirements when possible and applicable. In short, the separation-of-concerns principle has become (more than ever) a top priority.

A variety of software development approaches has been emerging, in a sense all focusing on achieving the separation-of-concerns principle. In the scope of this paper, we call these approaches feature-oriented software development practices, and we characterize this concept by referring to some prominent illustrations. For instance, Aspect-Oriented Software Development (AOSD) attempts to deal with the separation-of-concerns challenge by creating the notion of an aspect at all stages of the lifecycle of a software development process (e.g. [22, 3, 1]). From this perspective, an aspect can be defined to be a specific solution for a specific concern (category of requirements); and application assembly would ultimately be based on the composition of aspects that each meet a specific and coherent set of requirements. Collaboration-based development is in many ways similar to this approach: a software system is represented and based upon a core that supports the key (kernel) abstractions, and features are added to this core to deal with extra (functional or non-functional) requirements. This concept of collaboration-based design has become extremely relevant within the context of software product lines (e.g. [5]), but it is recognized to become essential for the maintainability of large and complex software applications in general. Hence our viewpoint that feature-oriented composition is a key concept for modern and future software development approaches.

Obviously, feature-oriented compositions must also support *dynamic* composition. Dynamic composition is required for dynamic modification of software systems, simultaneous (context-sensitive) modification of software systems for different clients (leading to multiple co-existing instances of the system) and distributed modifications that challenge the consistency of the operational software system.

Obviously, the notion of dynamic feature-oriented composition is an essential extension of the conventional object model[1] that was originally created to tackle software development from a programming (rather than a composition) oriented perspective. In this paper we define an enhanced object model with feature-based dispatch. This object model, which is named the Lasagne Object Model (LOM), is used as a basis for defining programming languages and component models that are well-suited to support feature-based composition. Articulating enhanced versions of the object model has been proven relevant when new dimensions of software construction have been explored[2, 11]. We believe that this type of research will remain crucial in creating a deeper understanding

[1]An object model describes the key concepts and mechanisms of a particular approach to object-oriented programming. It consists of a structural and a computational viewpoint. When referring to the *conventional object model* we mean the basic object model that extends the concept of abstract data types with polymorphic message invocations, classes and inheritance[41].

of the ongoing evolution in software development, and in creating the appropriate language or framework technology that has to support this evolution. In this particular context our enhanced object model supports feature-oriented composition.

Our paper is structured as follows. In section 2 we elaborate on the approach that we have followed. In section 3 we define the requirements for the Lasagne object model (LOM) in the light of the dynamics of feature-oriented composition that we have sketched above. In section 4 we present the LOM. In section 5 we provide a concrete programming language that implements the LOM. In section 6 we describe how the LOM can be structured as an incremental extension of the conventional object model. In section 7 we demonstrate the capabilities of the LOM in supporting consistent dynamic selection of features in a distributed environment. We evaluate the LOM and compare it with related work in section 8. We conclude in section 9.

2 Overview of our approach

Specifying composition in terms of features is often more powerful compared to specifying composition in terms of objects or classes. Most stakeholder who are involved in defining and constructing applications typically discuss and describe the application in terms of the features it supports. Therefore, we introduce the notion of a *feature identifier* which is an interpretable, high-level name that uniquely identifies the feature. The key idea behind the LOM is that various features can be added to a software application by specifying feature identifiers. These feature identifiers are used as run-time information that dynamically adds features, and hence influences the method dispatch process.

An important characteristic of our approach is that the LOM provides support for feature-based dispatch in distributed applications. More specifically, the LOM enables *remote selection* of features with the intention to support client-specific customization of distributed services . Clients simply select those features to be executed by attaching the corresponding feature identifiers to their client requests.

We now discuss a short example in which a basic library application is extended with two features. This pedagogical example will clarify the key characteristics and needs of the kind of object model we envisage. The library application assists the library personnel with the management of books and the process of loaning books to subscribed clients. Figure 1 shows a simplified structure of the application. It distinguishes between three features. The base feature is the minimal functional core of the library application, which is always executed. It consists of three classes: BookCopyComponent of which the instances represent a particular copy of a particular book, BookManagerComponent that represents the operation of the library and ClientComponent that offers a (user) interface to clients of the library.

Two features are behavioral extensions of the base feature. The `notification` feature is an implementation of the subject-observer design pattern[15]. The im-

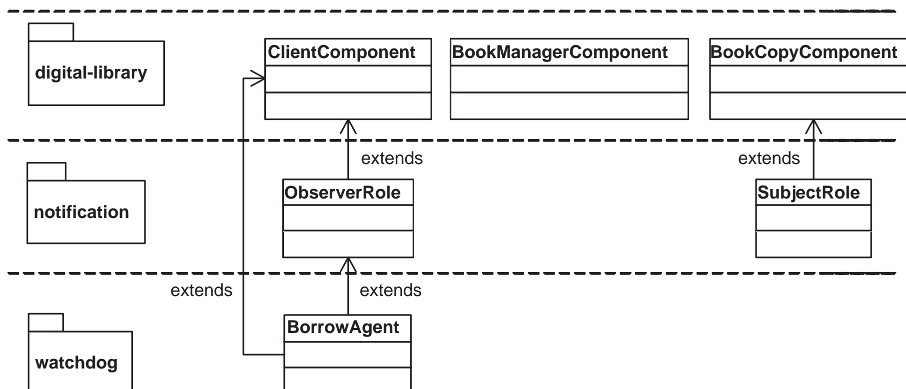

Figure 1: Decomposition of the library application in three features. The implementation of each feature is captured in a separate package.

plementation might be as follows. When a client has the intention to borrow a particular book that is momentary not available, it is registered as observer with the corresponding BookCopyComponent instance. When that bookcopy is then returned, a message is sent to the client. The second feature, named `watchdog`, extends the first feature such that the book copy, when returned, is automatically borrowed on behalf of the client.

In practice, features can be implemented as a set of class extensions, each particular member of the set extending a particular class of the given application. Many contemporary programming techniques would enable (albeit in some cases quite primitively) an implementation, for instance as a set of subclasses or mixins[8] in the case of object-oriented programming, or as a set of advices and inter-type declarations in the case of aspect-oriented programming[22]. However, what is really important is the question how feature identifiers can be represented in the program and how features can be dynamically selected by clients. As already indicated in figure 1, we use *name spaces* for packaging the implementation of each feature in separate identifiable units. For example, in Java each feature would be implemented in a separate package. The names of these packages can practically serve as feature identifiers.

Clients can then select features by attaching feature identifiers to their client requests. An intuitive example of a language construct that exactly does this is illustrated in figures 3 and 2: the `@`-construct.

Figure 2 shows the client-server architecture of the library application. In particular it shows the code of three client programs that execute at different hosts and use the services of the library application. To illustrate the semantics of feature dispatch, figure 3 interweaves the three client programs into a demo program to denote a particular sequence of execution at the server. The comments in the code describe what happens during execution.

In the code segments shown in figures 2 and 3, we have used the syntax

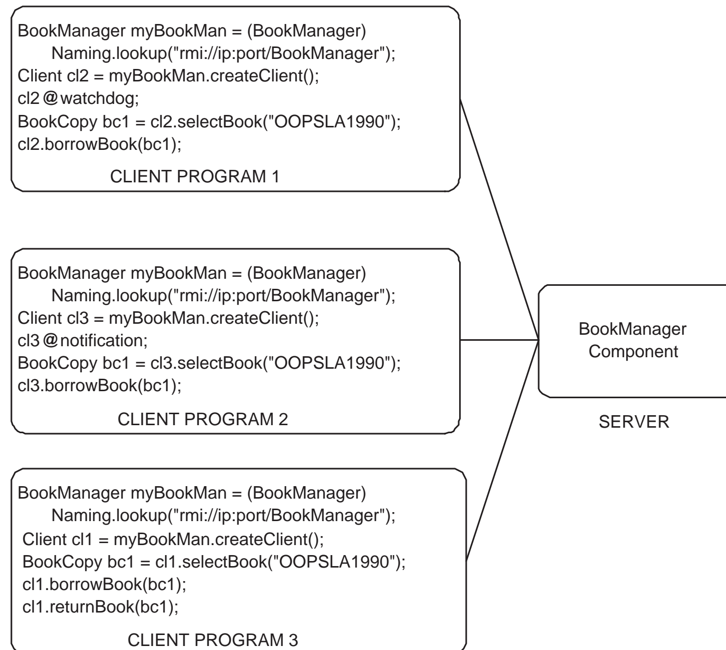

Figure 2: Three client programs use and customize the library application that runs on a separate server

```
<reference>"@"<feature identifier>;
```

to associate a specific feature (i.e. its identifier) to an object reference, thus expressing that the messages sent to the reference will be consistently processed by the specific feature implementation. We call such annotated references *type-annotated references* because the associated feature identifiers determine the dynamic type of the referenced object as observed by the holder/user of that reference. We can indeed state that the dynamic type of an object is determined by the annotated feature identifiers. For example, when in the program illustrated in figure 3, a message is sent to reference `cl3@notification`, the corresponding Client instance will execute the behavior of the ObserverRole class extension and any BookCopyComponent instances used in the subsequent control flow will execute the behavior of the SubjectRole class extension. Thus, the `@`-construct is used to annotate references with a dynamic type that corresponds with a particular selection of behavior.

The next section enumerates eight requirements that the LOM should meet in order to support feature-based dispatch in mainstream distributed software development.

```
import digital-library.*;
import java.rmi.*;

public class ThreeInterwovenClientPrograms {

 public Main() {
 BookManager myBookMan = (BookManager)
 Naming.lookup("rmi://ip:port/BookManager");

  Client cl1 = myBookMan.createClient();
     Client cl2 = myBookMan.createClient();
        Client cl3 = myBookMan.createClient();

     cl2@watchdog;
     //client-specific selection of the
     //watchdog feature and, implicitly,
     //the notification feature.

        cl3@notification;
        //client-specific selection of the
        //notification feature

 BookCopy bc1 = cl1.selectBook("OOPSLA1990");
  cl1.borrowBook(bc1);

     cl2.borrowBook(bc1);
     //notification will register cl2
     //as observer with bc1, because bc1 was
     //already borrowed by cl1

        cl3.borrowBook(bc1)
        //notification will register
        //cl3 as observer with bc1, because
        //bc1 was already borrowed by cl1

  cl1.returnBook(bc1);
  //bc1 will notify all its observers that
  //it has been returned. As a result watchdog
  //will try to borrow bc1 on behalf of cl2
 }
}
```

Figure 3: A demo program that illustrates how three client programs customize the library application in their own context without affecting each other.

3 Requirements

The example that has been sketched in the former section illustrates the needs for dynamic feature composition. This section enumerates the requirements that must be supported by an extended object model - LOM in fact - to support feature dispatch as a key element in developing software for distributed applications.

First of all, the object model should support *context-sensitive late binding*. To achieve context-sensitive modification of software systems we need to adjust the mechanism of late binding of self. To enable context-sensitive modification, objects are often incrementally modified with different variant implementations of the same base feature. When an operation of an object is invoked, we expect the methods of the right variant (that is the variant best suited for meeting

the context-specific requirements) to be executed. For example when different clients of the same system require different variants of a specific feature, the choice of the correct variant when dispatching a message depends on the requirements of the client sending that message. Notice that the late binding mechanism of traditional OOPL is always selecting the most recently defined variant in the class hierarchy that describes the actual object being addressed. Hence the method lookup process should not only be controlled by the receiver object, but should also be sensitive to properties of the usage context such as requirements of message-passing clients and to actual properties of the operating environment.

Second, opening up the method lookup process to message-passing clients should not violate *encapsulation*. The principle of encapsulation[34] states that changing the private implementation of an object without changing its public interface may not break client code.

Third, the object model should support *local behavioral consistency* in the dynamic selection of features. When a message-passing client selects a certain specialization of a method, subsequent self calls within the execution of the latter method should be consistently redirected to the enclosing class extension.

Fourth, context-sensitive selection of features should be supported in *distributed execution environments* such that clients can modify the behavior of (multiple) remote servants.

Fifth, the object model must still allow for *receiver-controlled late binding*. The conditional logic that determines whether a feature must be selected for a given message is not only determined at the message sending side but also at the message receiving side (i.e. the server-side). For instance, this is certainly the case for features such as authorization, persistence and replication. In short, both client objects and server objects must be able to control the use of specific features.

Sixth, the object model should support *large-scale features* that crosscut through multiple base classes. A feature implementation often cannot be captured in a single class extension, but is distributed across multiple class extensions. The feature dispatch mechanism should be able to support dispatching to such large-scale features without the need to refer to participating objects of that feature.

Seventh, given the large-scale nature of a lot of feature implementations, client-specific selection of features poses the question of *global combination consistency*: within a given client-specific view, all messages executed by an application, must be applied to the same combination of features. This means that context-sensitive late binding must be applied to the whole collective behavior of a collaboration between multiple objects, not just to single objects. A coordination mechanism is necessary to enforce the harmonious interpretation of self parameters within a client-specific view, such that the same combination of features is consistently applied.

Last but not least, the object model should be *generic*. Existing language constructs, such as delegation[24], qualified message passing[10] and predicate dispatching[14] already support context-sensitive late binding to a limited ex-

tent. It is clear that we aim for an object model that subsumes and extends such constructs in a distributed execution environment, instead of inventing similar constructs from scratch. Furthermore, a detailed study of the existing literature has revealed that some of these language constructs also expose significant problems[40]. So the required object should also reach beyond the limitations of the above mentioned language constructs.

Since our objective is to present a new object model for feature dispatch in distributed applications, it is obvious that our main priority is to bring together the above requirements in a coherent frame. When implementing the object model in a practical environment, performance is also a concern. It should be stated that an implementation of an object model for distributed systems has not the same kind of expectations as when the application area is a single virtual machine. After all, any temporal overhead in local method dispatch is negligible in comparison to the delays caused by excessive or wrong use of the network.

Having said this, the rationale for an enhanced object model stems from the lack of native language support for feature dispatch. Dynamic and context-sensitive composition of features is required in more and more applications. In a language, that does not natively support this, the programmer must design and implement it manually. This requires additional implementation effort, it is probably less efficient than dedicated language support, and the resulting application is much more difficult to understand and maintain. Thus, while we certainly want to facilitate an efficient implementation of the Lasagne object model, the goal of this paper is to demonstrate the value of the model in terms of productivity and maintainability.

4 The Lasagne object model

In this section we present the LOM. It consists of a structural and a computational viewpoint. The structural viewpoint explains the different elements of the model, such as objects, messages, classes; and how they relate to each other. The computational viewpoint describes the dynamic behavior of these elements such as method dispatch.

4.1 Structural viewpoint

4.1.1 Decomposition

The smallest unit of decomposition is the component, a class that is made more reusable by encapsulating it in two ways: the services it provides as well as its dependencies are specified in terms of *interfaces*. This means that clients of this class are restricted to the *provided interfaces*, whereas the class's own dependencies are specified exhaustively by its *receptacles* and its *expected interfaces*. The former are similar to associations in UML, the latter will be discussed in the context of composition of components.

A collaboration is a combination of components that cooperate to provide a certain feature. To denote this grouping, the components are all tagged with

the same *feature identifier*: a name for the collaboration. This identifier is also used to specify a collaboration's dependencies on other collaborations.

4.1.2 Composition

Composition is performed using object-based inheritance (also called delegation [24] or wrapping). A component A can be wrapped by another component B if A's provided interfaces matches B's expected interfaces. Since collaborations are aggregates of components, composition of collaborations can be seen as composition of their components.

An application[2] is built starting from an initial collaboration, the "stable core", which is then refined by collaborations that provide additional functionality. The stable core establishes the structure of the application, but provides only a bare minimum of functionality. The components that make up the stable core are called "core components", the components in the refining collaborations are known as "wrapper components".

4.1.3 Instantiation

Only core components are instantiated explicitly, wrapper components are instantiated implicitly. We will not go into details about the machinery behind this. For our purposes, it suffices to say that Lasagne provides flexible means to specify which wrapper components should be wrapped around a core component, and in which order.

The result of instantiating a core component which is wrapped by other components, is an aggregate object. Such an object is perceived as a whole from the outside – referred to as `self` from the inside. The constituent component instances can refer to themselves using `this`.

4.2 Computational viewpoint

4.2.1 Method dispatch

Besides grouping components into collaborations, feature identifiers are also used to achieve context-sensitive customization. To enable a subsystem to present itself to clients under different guises, message flow carries with it a set of feature identifiers that limits which collaborations should be considered in handling messages. This set is called the composition policy.

Clients change the composition policy by using a mechanism of annotation and interception. This mechanism consists of annotating a reference with a feature identifier f. This does not affect the current composition policy, but rather specifies that, when a method is invoked on this reference, f be added to that composition policy, and when the invoked method returns, f be removed. Thus the selection of f is only visible within the dynamic scope of the invoked method.

[2]An application is also called a "subsystem". Both terms refer to the whole of a stable core and the collaborations that extend it.

Another way of influencing method dispatch is by annotating methods with predicates. In contrast to the composition policy, predicate dispatching[14] enables the *receiver* of the call to influence dispatch. A predicate simply overrides the composition policy, without changing it. When a method does not have an associated predicate, this rule has no effect.

Contextual properties are another piece of information that is propagated along with the message flow. They can be used in an application-specific way to provide contextual information used in handling a message. They do not influence method dispatch directly, but client programs and predicates can evaluate over them. Unlike feature identifiers, they can be added and removed at any point in the execution.

When an eligible[3] message is processed by an object, the `self` parameter of that object is dynamically bound to a certain combination of components. What this combination exactly is, depends on the composition policy of that message, the predicates and the specified order of execution. The respective methods of the components are combined by means of a method combination parameter, called `inner`.

4.2.2 Combination consistency

Two rules ensure local and global consistency in the combination of features. The first and most important rule is that the composition policy must propagate with the (distributed) message flow. This means that in a control flow of consecutive messages, a composition policy is copied from the one to the next message, etc. As such, knowledge about the selected combination of features travels with the locus of execution. A composition policy is serializable, thus it is easy to achieve propagation with distributed message flow.

The second rule is about ensuring global combination consistency in the presence of callbacks. It states that the composition policy should 'stick' when an object passes itself as argument in a message to another object. After all, when the other object calls back, we want to select at least the same combination of features that were activated when the object has passed itself. This is achieved as follows. When an object passes itself, a new reference is created at the receiving object. The feature identifiers, which are then in the composition policy, must be transparently annotated to the newly created reference. Hence the expression that the composition policy should 'stick'.

The Lasagne object model as sketched above will be illustrated within the context of a practical programming language, in order to illustrate the above mentioned concepts in further detail.

5 A concrete programming language

In this section, we describe a concrete extension of the Java[29] programming language. This language mainly serves as a pedagogical vehicle to communi-

[3]having a matching selector

cate and illustrate the Lasagne Object Model. We refer to this language as the Lasagne programming language. It is of course just one possible approach to support the LOM. The LOM is actually the result of reflecting on two implementation projects[18, 39] in which two different object-oriented technologies have been extended with support for dynamic and context-sensitive composition of features.

The Lasagne programming language distinguishes between three phases: (1) component implementation, (2) component composition, (3) and context-sensitive selection of features. Every phase is supported by means of one or more language constructs. In the rest of this paper we present these language constructs. To keep the overview within certain spatial boundaries, this section only targets single-composed features and defers issues related to programming crosscutting features, support for remote customization and global combination consistency to section 7.

5.1 Implementation of components

As stated in section 4, an application is built starting from a stable core, which is then refined by additional collaborations. The stable core is implemented as a set of core components, whereas the additional collaborations are implemented as sets of wrapper components.

A core component is defined as a class that implements one or more interfaces. For example the core component Parent, shown below, implements the interfaces DeclaredParent and FooInterface.

```
package stablecorename;

public interface DeclaredParent {
  public void b();
}

public interface FooInterface{
  public void foo();
}

public abstract class AbstractParent
                    implements DeclaredParent {
                               FooInterface
  public void b() {...}

  public abstract void foo();
}

public class Parent extends AbstractParent {
  public void b() {
      ...
      super.b();

  }

  public void foo() { ... }
}
```

Wrapper components are implemented as classes that are declared to *wrap* instances of a given interface type. Like an `extends` clause to specify a superclass, a `wraps` clause is used to state the *static wrappee type*. This also declares

the wrapper component to be a subtype of the static wrappee type. For example, the wrapper component MyWrapper, shown below, declares to wrap the DeclaredParent interface.

```
package myFeatureIdentifier;

expects otherFeatureIdentifier;

import stablecorename.DeclaredParent;

public interface Adonis {
   public void bar();
}

public class MyWrapper wraps DeclaredParent
                    implements Adonis {

   public void bar() {...}

   public void b() {...}
}
```

Components are labelled with their unique feature identifier by means of the `package` statement. As such we use conventional package names for defining feature identifiers. For example, MyWrapper is declaratively bound to feature identifier "myFeatureIdentifier", and Parent is declaratively bound to feature identifier "stablecorename".

The `expects` statement is used to indicate dependencies between collaborations. For example MyWrapper declares that it depends upon another collaboration with feature identifier "otherFeatureIdentifier".

Wrapper components can either specialize methods of the wrappee or add new methods that do not exist in the wrappee. Adding new methods is simply done by implementing a new interface. For example MyWrapper implements a new interface, called Adonis, which declares the operation bar().

Specializing methods override an existing method of another component (from another collaboration). They may refer to the wrappee by means of the keyword `inner`. They must use this keyword to forward the original call to the wrappee. For example, method b() of MyWrapper is a specializing method and might look as follows:

```
public void b() {
      ...
      inner;
      ...
}
```

Finally, note that the use of class-based inheritance is not forbidden; components may internally be structured as a class hierarchy. For example, class Parent extends from AbstractParent.

5.2 Composition of components

An instance of MyWrapper must always wrap an instance of DeclaredParent or a subtype thereof to assure that forwarding of calls via `inner` never fails. Such correct composition is assured by specifying a so-called *composition pattern.*

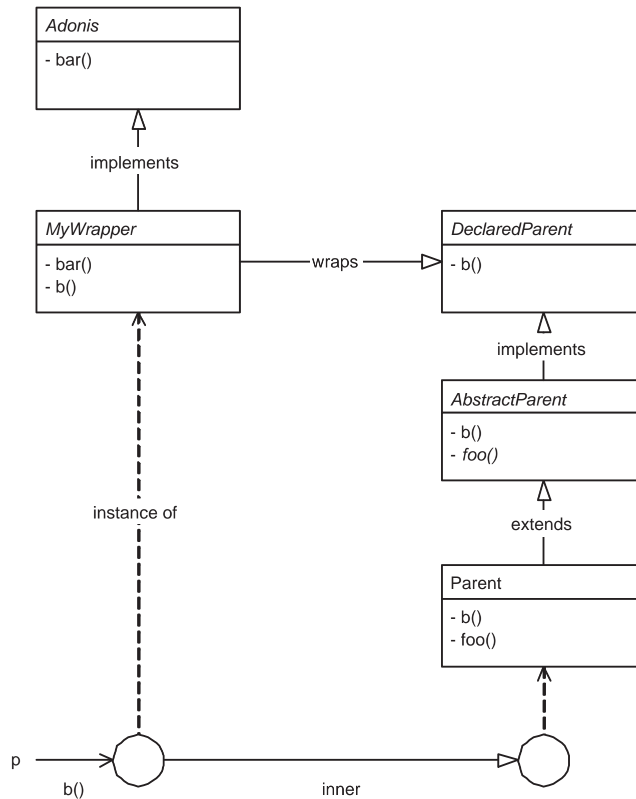

Figure 4: Since the @-construct has been applied to reference p, the behavior of class MyWrapper is activated for messages that are sent via p.

A composition pattern specifies an ordered set of wrapper components around the core component. It is important to understand that a composition pattern does not lead to the automatic creation of a new, composite class, as would be the case in mixin-like composition. A composition pattern merely specifies a range of allowed component configurations and what are the allowed orderings of execution[4]. For example, the declaration

```
composition Parent = MyWrapper<Parent>;
```

states that MyWrapper *may be* composed with Parent and that when MyWrapper's feature identifier is effectively selected by a client, the methods of MyWrapper must be executed before the methods of Parent.

Composition patterns are specified in the beginning of the main program that bootstraps the application.

[4]Overriding precedence is thus implicitly specified by the ordering in the composition pattern.

5.3 Context-sensitive selection of features

To activate or deactivate a feature we introduce the following language constructs:

5.3.1 The @-construct

To effectively activate the behavior of a feature, the programmer must use the @-construct that was already shortly introduced in section 2. Thus, instances of wrapper components are never explicitly created by means of a constructor call, but they are implicitly created as part of executing the @-construct. For example, the program below shows how the behavior of MyWrapper is effectively activated for a particular Parent object:

```
DeclaredParent p = new Parent(...);

p@myFeatureIdentifier;

p.b()
//executes MyWrapper.b();
```

Figure 4 shows a class diagram[5] that graphically represents the program listed above.

Each time the @-construct is applied to a reference, the underlying language runtime associates the reference to the passed feature identifier. Then, for every message sent along a reference, the feature identifiers associated to that reference will be added to the composition policy of that message.

5.3.2 The #-construct

Feature identifiers may be subsequently deactivated by using another language construct: `<reference>"#"<feature identifier>`. The following program that is a continuation of the above program illustrates this:

```
p#myFeatureIdentifier;

p.b();
//executes Parent.b();
```

Here, the application of the #-construct removes the association between reference p and "myFeatureIdentifier". As a result, messages that are subsequently sent via p will not be dispatched to MyWrapper anymore.

5.3.3 Constructive cast operator

Wrapper components may also implement new methods that are not found in the core component. This was already illustrated by the above MyWrapper class that implements the Adonis interface. The program below shows how to invoke such add-on methods.

[5]The graphical notation of the figure is due to [23].

```
DeclaredParent p = new Parent();

//p.bar();
//compile time error

Adonis c = (Adonis)p;

c.bar();
//executes MyWrapper.bar();
```

The p reference must first be casted to the Adonis interface. Since the p reference points to an instance of Parent, the cast to Adonis would seemingly fail. This is however not the case. In fact, the cast to Adonis actually is a new language construct that is called the *constructive cast operator*. Its syntax is as follows: `(<interface type>)<reference>`. When applied to the specified reference, the constructive cast operator returns a new reference to the component instance that implements the specified interface. So the statement

```
Adonis c = (Adonis)p;
```

assigns the MyWrapper instance, which is wrapped around the p instance, to reference c. Furthermore, the feature identifier of MyWrapper is also transparently associated to the reference c. Notice that the constructive cast operator cannot be used for casting to class types, only to interface types.

5.3.4 Predicate dispatching

The above language constructs support sender-initiated late binding. There is however also need for receiver-controlled late binding to enforce application-specific feature selection logic upon all clients. To support this, the Lasagne programming language supports a variant of predicate dispatch[14]. Predicate dispatch is a generic method dispatch mechanism that allows to control the applicability of methods by means of logical predicates. A method is applicable (i.e. must be executed for the processing of a received message) when the associate predicate is momentary true. A predicate can evaluate over the values and types of the actual message parameters, the state of the object, etc.

The Lasagne programming language takes over this technique[6] by introducing the `when` clause. For example suppose that the MyWrapper component is revised as follows:

```
public class MyWrapper wraps DeclaredParent {
   private int counter = 0;

   public void b() when (counter < 1)
   {
     counter++;
     ...
     inner;
     ...
   }
}
```

[6]The logical theory described in [14] to determine overriding precedence between methods is not applicable since in Lasagne predicates evaluate over values that are defined during run-time. In Lasagne overriding precedence is explicitly specified by means of composition patterns.

Method b() is associated to a predicate expression that specifies that b() is only applicable when the value of the counter instance variable is less than 1. As a result, in the execution of the following program, method b() of MyWrapper will be executed the first time a message b() is sent to object p, but not the second time:

```
DeclaredParent p = new Parent();

p.b();
//MyWrapper.b() is executed

p@myFeatureIdentifier;

p.b();
//Parent.b() is executed.
```

It may seem odd that method b() has not been executed the second time. After all at that moment "myFeatureIdentifier" was activated in the current composition policy. However, the point is that predicate expressions are able to override the composition policy. In other words, predicate expressions have precedence over the sender-initiated selection logic and, therefore, are able to enforce application-specific conditional logic upon all clients.

When no predicate expression are associated with a method, applicability of this method is again under full control of the composition policy.

5.3.5 Delegation

Delegation[24] is a technique that allows sharing of behavior at the level of objects. An object, called child, can have references to other objects, called parents. A message for which the message receiver has no applicable method is automatically forwarded to one of its parents. When in the class of the parent an applicable method has been found, the self parameter of the parent object will be automatically bound with the message receiver[23].

The LOM supports a special form of delegation, namely single delegation which is delegation to a single parent. For example, suppose that the class MyWrapper has been implemented as shown in the paragraph on local predicate dispatching and that the implementation of class Parent is as follows.

```
public class Parent extends AbstractParent {

  public void b() {...}

  public void foo() { self.b(); }
}
```

Then in the program below, the self call to b() during the first execution of method foo() is dispatched to method MyWrapper.b(), while the self call during the second execution is dispatched to method Parent.b().

```
DeclaredParent p = new Parent();

p@myFeatureIdentifier;

((FooInterface)p).foo();
//self.b() in method foo() is
//executed by MyWrapper.b()
```

```
((FooInterface)p).foo();
//self.b() in method foo() is
//executed by Parent.b();
```

This example demonstrates the `self` parameter is dynamically bound to a certain combination of component instances on per message basis. What this combination exactly is, depends on the composition policy and the predicate expressions.

5.3.6 Controlled delegation

Pure delegation is a very dangerous technique because, similar to class-based inheritance, it introduces several problems related to the specialization interface (such as encapsulation violations and broken contracts). Delegation is however much more dangerous than class-based inheritance because of its flexibility. After all, it is well-known that with pure delegation, anyone who can access the client interface of an object can also see its specialization interface. Thus, delegation enables an object to extend any other object. The consequence of this is that in large applications the problems related to specialization become unmanageable because it becomes very difficult to localize software defects due to these kinds of problems. Steyaert et. al [36] has already studied this problem and proposes as solution that a parent *controls* which objects may delegate to that parent. This makes it much more easier to localize and resolve software defect related to encapsulation violation and broken contracts. After all, one statically knows the complete delegation hierarchy of a parent object and therefore one can easily inspect the overall situation to trace down the cause of a software defect.

Lasagne has a similar mechanism. Here, composition patterns determine which components may be composed together into one object and it is only within the boundaries of such an object that delegation relationships are allowed. This restriction does not conflict with sender-initiated late binding. After all, message senders can still influence the self parameter of an object by dynamically selecting between the allowed set of components in that object.

5.3.7 Consultation

Even in a controlled setting, late binding of self across the component instances is not always desired because it does not fit with the requirements of the particular application. For some self-referential calls, it is not all desired that these calls are dispatched to other component instances. Instead, it is preferred that these calls are dispatched to the methods of the static class hierarchy of the calling component.

Lasagne supports this by introducing the second self-referential parameter: `this`. For example, suppose the Parent class is programmed as shown below. Then every `this` call will be executed by Parent or its superclass. In other words, here we just have conventional late binding in class-based hierarchies.

```
public class Parent implements DeclaredParent {
                               FooInterface
  public void b() {...}

  public void foo() { this.b(); }
}

DeclaredParent p = new Parent();

p@myFeatureIdentifier;

((FooInterface)p).foo();
//this.b() in method foo() is
//executed by Parent.b()
```

5.3.8 Feature identifiers encapsulate class names

It is important to understand that feature identifiers are part of a component's interface. This becomes clear when comparing the `@`-construct with qualified message passing[10]. Qualified message passing offers at first sight an easy tool for expressing sender-initiated late binding at the object level. Simply qualifying the message with the names of a class allows to specify which combination of methods in the class hierarchy must be selected.

However, qualified message passing violates encapsulation by exposing the inheritance structure of each class. For example, consider a class A that inherits from a class B, and a client class X that sends messages, qualified with class name B, to instances of A. In this setting class X is aware that A inherits from B. This very fact violates the encapsulation principle because if we changed the implementation of class A such that it inherits from a class D instead of a class B, then the implementation of the client class X would break.

Lasagne abolishes the pure form of qualified message passing. After all Lasagne enforces a component-oriented programming model. As such, message-passing clients can only refer to an object by means of an interface type and never by a class type. However, the semantical equivalent of qualified message passing is simply expressed by annotation of feature identifiers. The point we want to make is that message-passing clients refer not by class name to a specific component of an object, but by feature identifier. Thus, feature identifiers *abstract away the class types of components*. This also means that feature identifiers belong to the interface of a component and therefore feature identifiers cannot be changed. This is acceptable because feature identifiers are of a sufficient abstraction level.

5.3.9 Type transparency

The Lasagne programming language also provides support for type transparency[9]. This means that wrapper components are not only of the static, but also of the actual wrappee type. For example, a `MyWrapper` instance wrapping a `Parent` instance is also of the latter type and not just of type `DeclaredParent`. Hence, such an aggregate can be assigned to a variable of type `Parent` and the latter's

methods can be called on it. For example, in the following program fragment the cast to FooInterface succeeds[7]:

```
DeclaredParent p = new Parent(...);
p@myFeatureIdentifier;
((FooInterface)p).foo()
//executes Parent.foo();
```

It succeeds because when the @-construct is applied to reference p, the latter is not switched to the MyWrapper instance. Instead, the "myFeatureIdentifier" feature identifier, passed with the @-construct, is associated to the reference. Although "myFeatureIdentifier" is now activated, the MyWrapper instance will be skipped because it has no matching signature for the message foo().

6 Integration with the conventional object model

In this section we show that the Lasagne object model can be considered as an incremental extension of the conventional object model. Incremental extension means that we cam extend the conventional object model without invasively changing its concepts. As such, it becomes easier to implement the Lasagne object model as a modular plug-in of existing object-oriented programming languages.

First we will give an overview of the conventional object model. Then we present the Lasagne object model as an incremental extension of the conventional object model.

6.1 The conventional object model

We use a semi-formal notation to describe the conventional object model. This enables us to explain the relation with the LOM as precisely as possible. The formal notation as well as the description itself is largely based on [7].

In traditional object-oriented programming an object can be represented as follows. Let $OBJECT$ be the set of objects, L be the set of all selectors, MSG the set of all messages and $reject$ the non-existent element. Then, an object $o \in OBJECT$ is defined as $o = (I, M, S, P, R, value, this)$ where $I \subset L$ indicates the interface of the object, M the set of methods, S the state space formed by the instance variables, P the mapping from the interface to the methods, R the set of references to acquaintance objects of o, $value$ the mapping between these references and specific objects, and $this$ the self-referential pseudo-variable.

(i) The interface of the object is the set of selectors that the object can respond to. Thus, $I = \{selector(m) | m \in M\}$ where $selector : M \to L$ is a function that returns the selector by which a method $m \in M$ can be invoked.

[7]Since component-oriented programming is the basic starting point of our work, messages may only be sent to references with an interface type.

(ii) The mapping P is defined as $P : L \rightarrow M \vee reject$, i.e. a selector is either mapped to a method in the method set or rejected [7].
(iii) A reference $r \in R$ is defined as $r = name$ where $name$ defines the variable that is declared by the object's class.
(iv) The associated reference mapping $value : (R \cup \{this\}) \rightarrow OBJECT$ retrieves the value of references, i.e. $\forall r \in R : value(r) = q \in OBJECT$ where q is the object to which reference r points.
(v) The self-referential pseudo-variable $this$ is defined as a reference that gives o access to its own state and behavior, i.e. $value(o.this) = o$.
Note that once the object o is instantiated from its class, the value of $this$ is completely fixed and cannot be extended during the entire object's lifetime.
(vi) Objects interact by sending messages. The notion of a message is defined as $e = (s, n, l)$ where s is the sender object of the message, n is the receiver object and $l \in L$ is the selector[8].
(vii) Finally we define the behavior of objects. The behavior of an object is visible by executing the methods of its class (i.e. when an object receives a message, the message is dispatched to a method $m \in M$ by means of the mapping P). Formally, we define the behavior B of an object $o = (I, M, S, P, R, value, this)$ as

$$B(o) = \{b_m : S \rightarrow (\{(o, n, l) \in MSG | \ n \in value(R \cup \{this\}) \}, S') \mid m \in M \}.$$

The behavior b_m of a method m executed at an object o is defined as the state change of o and one or more message sends from o to other objects represented by the union of the references R and the self-referential pseudo-variable $this$ of o. In this simple notation, we do not take into account state changes at other objects or at object o itself due to messages sent by b_m.

6.2 Formulating the delta

We now give an informal description of how the LOM can be structured as an incremental extension on top the conventional object model. Figure 5 gives a high-level overview of this.

We first discuss how the concepts of the conventional object model fit into the LOM. Subsequently we overview how the key concepts of the LOM can be integrated on top of the conventional object model.

6.2.1 Relation to the conventional object model

An object o consists of an ordered set of component instances ($< co_o$, co_1, ..., $co_k >$. Each component instance is very similar to the conventional object model that has been introduced in the previous section. In other words, each component instance has an interface I, a set of methods M, a state space S, a mapping P, a set of references R, a reference mapping $value$, and a self-referential parameter $this$. The semantics of these elements are very similar to the conventional object model.

[8]We do not take into account message parameters in this notation.

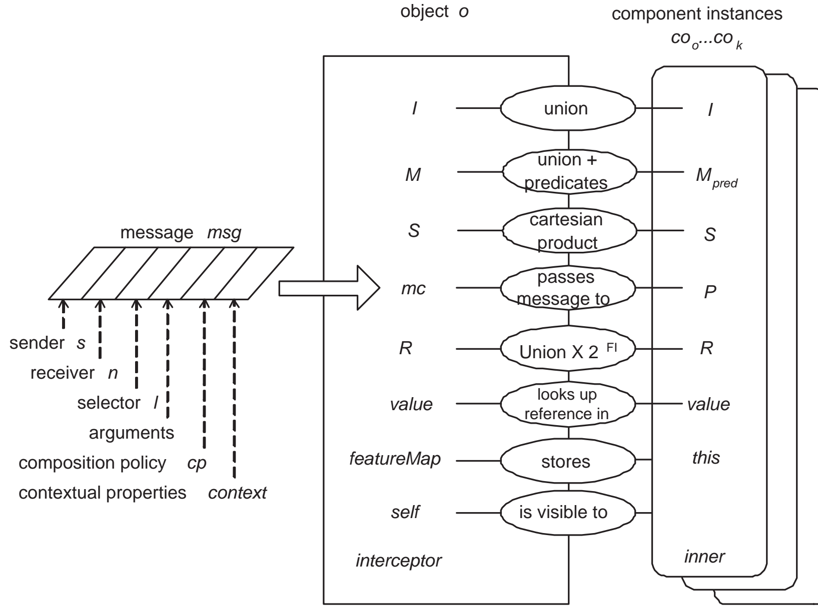

Figure 5: A high-level overview of the Lasagne object model

The behavior, state and structure of all these component instances constitute the behavior, state and structure of the overall object o. In other words, the object o has an interface I, a set of methods M, a state space S, a set of references R, a reference mapping $value$; and all these elements are defined in terms of the corresponding elements of o's component instances. To distinguish the elements of o from those of a component instance co_i, we refer to the former as $o.I$, $o.M$, etc. and to the latter as $co_i.I$, $co_i.M$, etc.

6.2.2 Adding the key concepts

Figure 5 shows how the Lasagne object model can be integrated with the conventional object model. It introduces new concepts at the level of message structure, at the level of the whole object o and at the level of the component instances of o:

Message structure The traditional message structure is extended with the composition policy cp and a set of contextual properties $context$.

Object At the level of the object o, five new concepts are introduced:

- the mapping $featureMap$ manages the mapping between feature identifiers and component instances.
- the self-referential parameter $self$. The evaluation of $self$ is the central element of the Lasagne object model because it defines the

basic semantics of the context-sensitive method dispatch mechanism of the Lasagne object model. The value of $self$ is, as opposed to *this*, not fixed at runtime, but it is dynamically evaluated on a per message basis using the composition policy and predicates.

- *mc* represents the method dispatch mechanism of o. It passes every received message to the appropriate component instance's mapping P which will in turn map the message to the appropriate method. *mc* uses the value of $self$ to know to which component instance it must dispatch.
- For each reference $r \in co_i.R,\ i : o..k$, $o.R$ keeps the mapping between r and its annotated set of feature identifiers $\{f_1, ..., f_q\} \in 2^{FI}$.
- *interceptor* provides the basic mechanism for updating the composition policy of messages that are sent by o. It consults $o.R$ to know which feature identifiers must be added to which message.

Component instances Each component instance has besides the *this* parameter also access to the $self$ parameter of the enclosing object o, and to the method combination parameter, *inner*. Methods of a component instance use the $self$ parameter to refer to the whole object o, while they use the *inner* parameter to forward a received message to a compatible method of another component instance of o. Like $self$, the value of the *inner* parameter is dynamically evaluated on per message basis. Finally, predicates may be associated to specific methods of specific components. Predicates can evaluate over the state of the component instance and the received messages (including the composition policy and contextual properties associated with these messages).

A formal treatment of the Lasagne object model is elaborated in the appendix of this paper.

7 Feature dispatch in distributed applications

In this section we describe how the Lasagne programming language supports consistent dynamic selection of features in a distributed system. Special attention is paid to the following topics:

- Support for programming crosscutting features.
- Increased reusability of feature implementations by applying the principles of framework-based design and incorporating support for metaprogramming.
- Demonstrating the use of contextual properties.
- Support for global combination consistency in distributed execution environments and in the presence of callbacks.

7.1 Feature implementation

To demonstrate how crosscutting features can be programmed in Lasagne, we revisit the library case as introduced in section 2. The implementation of the subject-observer design pattern as presented in that section is not reusable since it cannot be composed with other applications than the digital-library application.

In this section we demonstrate how reusable feature implementations can be achieved in Lasagne. There are two mechanisms for this. The first mechanism consists of support for framework-based design. This means that we separate the reusable part of the subject-observer design pattern into an *abstract collaboration* and the application-specific part of the notification feature into a concrete collaboration.

The second mechanism is support for meta-programming[19]. As opposed to base-level programming, meta-programming enables implementing behavior in a highly reusable and versatile manner, but it disables type checking.
The Lasagne programming language offers a very specific meta-programming model, namely specializing methods that can be combined with all kinds of method signatures. We call such specializing methods *multi-signature methods*. To distinguish such multi-signature methods from ordinary methods, the Lasagne programming language imposes a simple signature convention: multi-signature methods must declare one parameter of the pre-defined interface type `Interaction` and the common superclass for objects as the return type. The name of the method can be chosen freely by the programmer.

```
public Object <name of method>(Interaction msg)
```

The `Interaction` parameter represents the original message that has been received by an object. It gives access to various properties of the message such as the selector of the invoked operation, and the parameter values of the message.

7.1.1 Abstract collaborations

A reusable implementation of the Subject-Observer pattern can be built as an abstract collaboration. This abstract collaboration is shown in figure 6. It consists of two components, respectively called ObserverRole and SubjectRole. The feature identifier of the abstract collaboration is defined as “reusable-subject-observer”.

ObserverRole implements interface Observer. This interface defines the following operations: start() for associating the observer to a specified subject, stop() for deregistering the observer from a specified subject, and update() which must be used by subjects to notify the observer. Listing 1 shows the implementation of ObserverRole. It is declared as an abstract class. Listing 1 and figure 6 show that start() and stop() are implemented by invoking the appropriate operation of the specified subject. Method update() is left abstract. The specific implementation of this method depends after all on the concrete application in which the reusable-subject-observer collaboration is used.

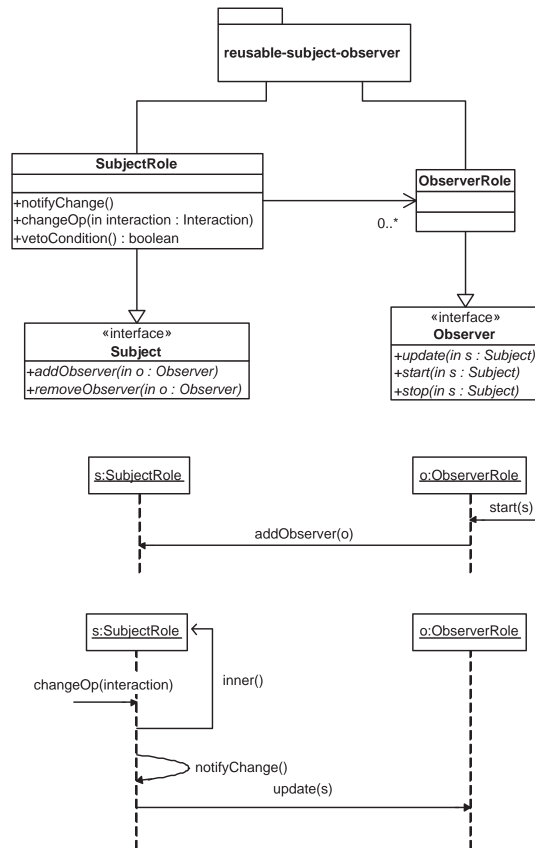

Figure 6: The reusable-subject-observer collaboration

Listing 1: The ObserverRole abstract component

```
package reusable-subject-observer;

public abstract class ObserverRole
                implements Observer {

   public ObserverRole() {}

   public abstract void update(Subject s);
   //this method is called by Subject instances

   public void start(Subject s) {
     s.addObserver(self);
   }

   public void stop (Subject s) {
     s.removeObserver(self);
   }
}
```

Listing 2 shows the implementation of the SubjectRole component. It implements the interface Subject. This interface defines operations for registering

and deregistering observers. SubjectRole implements these operations by storing observers in an instance variable of type List.

7.1.2 Multi-signature methods

SubjectRole also defines a multi-signature method, called changeOp(). It represents those operations of the application whose execution causes an event that observers must be notified of. The changeOp() method realizes the notification behavior by invoking the notifyObservers() method after the execution of such an operations returns. By implementing it as a multi-signature method, the code for the notification behavior can be implemented once and must not be replicated for every event-causing operation.

Listing 2: The SubjectRole abstract component

```
package reusable-subject-observer;

public abstract class SubjectRole
                implements Subject {
  protected List observers;

  public SubjectRole() {
     observers = new LinkedList();
  }

 public void addObserver(Observer o) {
     observers.add(o);
 }

 public void removeObserver(Observer o) {
     observers.remove(o);
 }

 public void notifyChange () {
    Observer elem;
    Iterator obs = observers.iterator();
    while (obs.hasNext()) {
      elem = (Observer)obs.next();
      elem.update(self);
    }
 }

 abstract public boolean vetoCondition();

 public Object changeOp(Interaction interaction)
                    when (!vetoCondition()) {
        Object result=inner;
        self.notifyChange();
        return result;
 }
}
```

For the rest, the changeOp() method is implemented very similar as specializing methods. It combines its behavior with the originally invoked operation by means of the `inner` keyword, and it is associated with a predicate as well. This predicate specifies when notification should not proceed. It makes use of a separate method vetoCondition(). This method is also left abstract because its concrete implementation depends on the concrete application at hand. For example an application-specific specialization of vetoCondition() could be that

if the same event occurs twice without any other events occurring in between, then a notification is only sent for the first occurrence.

7.1.3 Integrating the abstract collaboration

Since the reusable-subject-observer collaboration is abstract, it must be refined with a concrete collaboration that fills in the abstract methods with application-specific behavior. This section illustrates how the reusable-subject-observer collaboration is integrated with the library application by means of such concrete collaboration.

First we overview the stable core of the library application in more detail. As already stated in section 2, it consists of three core components: BookCopy-Component, BookManagerComponent and ClientComponent. Each component implements a separate interface. Listing 3 shows these interfaces. The semantics of the operations in these interfaces is straightforward.

Listing 3: The stable core of the library application

```
package digital-library;

public interface BookCopy {
  public void borrowIt()
      throws AlreadyBorrowedException;
  public void returnIt();
  public String getTitle();
  public boolean isBorrowed();
}

public interface BookManager {

   public Client createClient();
   //creates an instance of Client

   public BookCopy create BookCopy();
   //creates an instance of BookCopy


   public List books();
   //returns the list of books that
   //is offered by the library

   public void buy(BookCopy bc);
   //adds a specified book copy to the
   //list of offered books

   public void drop(BookCopy bc);
   //drops specified book copy from list
   //of offered books

   public BookCopy getBookCopy(String title);
   //returns a bookcopy that holds the specified
   //argument as title

   public void bookBorrowed(BookCopy bc);
   //notifies the book manager that the
   //specified book copy has been lent
   //to client

   public void bookReturned(BookCopy bc);
   //notifies the book manager that the
   //specified book copy has been returned
   //by a client
```

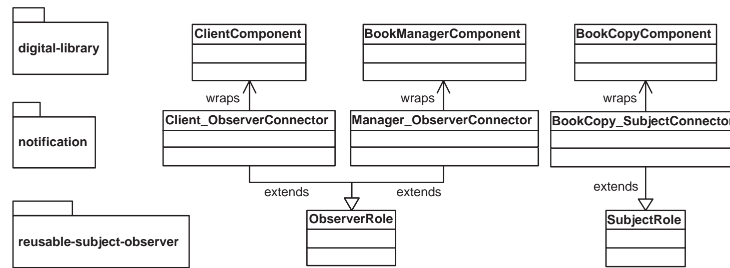

Figure 7: The notification collaboration integrates the reusable-subject-observer collaboration into the library application

```
}

public interface Client {
    public BookCopy selectBook();
    public boolean borrowBook(BookCopy bc);
    public void returnBook(BookCopy bc);
}
```

Now we integrate the reusable-subject-observer collaboration with the library application. A concrete collaboration with feature identifier "notification" is used to integrate the reusable-subject-observer collaboration with the library application. It consists of three wrapper components:

- the Manager_Observer_Connector wrapper, shown in listing 4, that connects the ObserverRole component to BookManagerComponent instances,
- the BookCopy_Subject_Connector wrapper, shown in listing 5, that connects the SubjectRole component to BookCopyComponent instances,
- the Client_Observer_Connector wrapper, shown in listing 7, that connects the ObserverRole to ClientComponent instances.

Figure 7 presents the resulting software structure. It is clear that components of the notification feature play the role of a middleman that adapts between the components from the other collaborations. The figure further indicates that the components of notification and reusable-subject-observer are composed by means of class-based inheritance. This is indicated by the `extends` clause. Inheritance is opted because the components of the reusable-subject-observer collaboration are abstract and, therefore, they cannot be instantiated nor composed at runtime. On the other hand, the notification collaboration and the digital-library stable core are composed by means of delegation. This is indicated by the `wraps` clause. Delegation is necessary because the notification collaboration is an optional feature that is only dynamically selected by clients.

As opposed to the example in section 2, the BookManagerComponent is also composed with the ObserverRole by means of the Manager_ObserverConnector class, which is shown in listing 4. It realizes the following behavior. When the BookManager instance buys a BookCopy, it is registered as observer with

that BookCopy instance. Then, when a BookCopy instance is lent to a client, the bookBorrowed() method will be invoked. Likewise, when that BookCopy instance is returned, the bookReturned() method will be invoked. The BookManager is deregistered after dropping the BookCopy.

Listing 4: The Manager_ObserverConnector wrapper

```
package notification;

import digital-library.BookManager;
import reusable-subject-observer.ObserverRole;

public class Manager_ObserverConnector
                extends ObserverRole
                wraps BookManager {

  public Manager_ObserverConnector() {}

  public void update(Subject s)  {
     BookCopy bc =  (BookCopy)s;
     Selector sel = (Selector)
       context().getProperty("InvokedChangeOp");
     if (sel.getMethodName().equals("borrow")) {
        ((BookManager)self).bookBorrowed(bc);
     }
     if (sel.getMethodName().equals("returnIt")) {
        ((BookManager)self).bookReturned(bc);
     }
  }

  public void buy(BookCopy bc) when true {
     inner;
     Subject bcs = (Subject)bc;
     super.start(bcs);
 }

 public void drop(BookCopy bc) when true {
      super.stop((Subject)bc);
      inner;
 }
}
```

Use of the constructive cast operator. Listing 4 also illustrates the use of the constructive cast operator. When messages cross collaboration boundaries, receiver object and parameter values must have types that are expected by the invoked method. For example, method buy() from Manager_ObserverConnector invokes the start() operation of the Observer interface and passes by reference the BookCopy component instance bc as parameter. Of course, since start() expects a parameter of type Subject, the bc reference must first be casted to the Subject interface. Since the bc reference points to an instance of BookCopyComponent, the dynamic type of bc is not a subtype of Subject. As a result, the cast to Subject seemingly fails. Here is where the constructive cast operator comes into play since it returns a reference to the component instance that implements the Subject interface.

Use of contextual properties. Furthermore, the Manager_ObserverConnector must dynamically map the update() operation of ObserverRole to the operations of BookManager: when borrowIt() is invoked on a BookCopy instance, update() must map to the bookBorrowed() operation; when returnIt() is invoked on a

BookCopy instance, update() must map to the bookReturned() operation. The mapping depends thus on the particular operation of BookCopy that was invoked within the current execution context. In other words, the mapping must be computed at runtime by taking into account message flow history of the currently ongoing computation. In the particular example, this is supported by representing the operation invoked on a BookCopy instance as a contextual property with name "InvokedChangeOp".

The BookCopy_SubjectConnector wrapper, shown in listing 5, defines the value of this contextual property. This is done in the overriding method changeOp() that retrieves the selector of the actually invoked operation (either borrowIt() or returnIt()).

Listing 5: The BookCopy_SubjectConnector wrapper

```
package notification;

import digital-library.BookCopy;
import reusable-subject-observer.SubjectRole;

public class BookCopy_SubjectConnector
                    extends SubjectRole
                    wraps BookCopy
{

   public BookCopy_SubjectConnector () {}

   public boolean vetoCondition() {
       return false;
   }

  public Object changeOp(Interaction interaction) {
      context().addProperty("InvokedChangeOp",
                    interaction.getSelector());
      return super.changeOp(interaction);
   }
}
```

Use of predicate dispatching. Finally note that all predicate expressions of Manager_ObserverConnector and the BookCopy_SubjectConnector evaluate to true. This means that the "notification" collaboration is always activated for the BookManager instance. On the other hand, there are no predicate expressions specified for the methods of the Client_ObserverConnector, which means that it is up to the client program to explicitly activate the "notification" feature for its interactions with the library.

7.1.4 Implementing a specializing collaboration

Now we discuss the implementation of the "watchdog" feature. The functionality of this feature is as follows. When a client wants to borrow a book that is already borrowed by someone else, a borrowing agent is activated that registers itself with the corresponding BookCopy instance as observer. When that bookcopy is returned the borrowing agent will deregister itself and will automatically try to borrow the book. The above steps are repeated until the book has been successfully borrowed by the client.

Listing 6: The "watchdog" collaboration

```
package watchdog;

expects notification;

import digital-library.Client;
import reusable-subject-observer.Observer;

public class BorrowAgent wraps Observer, Client {

  public BorrowAgent() {}

  public void update(Subject s) {
    BookCopy bc = (BookCopy)s;
    Selector sel = (Selector)
      context().getProperty("InvokedChangeOp");
    if (sel.getMethodName().equals("returnIt")) {
       Observer o = (Observer)self;
       o.stop(s);
       ((Client)self).borrowBook(bc);
    }
  }
}
```

The actual implementation partially reuses the "notification" collaboration. The code in listing 6 shows that the collaboration consists of a single wrapper component that reuses the borrowBook() and selectBook() methods and overrides the update() method of the Client_ObserverConnector component of the notification collaboration. This implementation dependency is explicitly specified by means of the `expects` construct. As a result, when a client selects the "watchdog" feature, the "notification" feature will be implicitly selected. This becomes clear when looking at the `self` call to borrowBook() in the update() method: this `self` call will always be redirected to the borrowBook() method of Client_ObserverConnector of the notification collaboration.

For the sake of completeness, the implementation of the Client_Observer-Connector component is shown in listing 7.

Listing 7: The Client_ObserverConnector wrapper

```
package notification;

import digital-library.Client;
import reusable-subject-observer.ObserverRole;

public class Client_ObserverConnector
                        extends ObserverRole
                        wraps Client
{
  public Client_ObserverConnector() {}

  public void update(Subject s) {
     ...
  }


  public BookCopy selectBook() {
      BookCopy bc = inner;
      if (bc.isBorrowed()) {
         Subject subject = (Subject)bc;
         super.start(subject);
      }
      return bc;
```

```
  }

  public boolean borrowBook(BookCopy bc) {
          Boolean success = (Boolean) inner;
          if (success.booleanValue())
             super.stop((Subject)bc);
          else
             super.start((Subject)bc);
          return success.booleanValue();
  }
}
```

7.2 Composition patterns

As already explained in section 4, composition of collaborations is specified by means of composition patterns that are specified per core component. Figure 8 shows the compositional logic for the library application. Note that this composition logic is specified at the server-side in the main program that bootstraps the library application. A construct, not mentioned before, is the `<-` construct by means of which multi-signature methods are bound to the appropriate selectors. For example, the changeOp() method is bound to the borrowIt() and the returnIt() operations of the BookCopy Interface.

Listing 8: Composition patterns are specified in the main program that bootstraps the library application

```
import digital-library.*; import notification.*;
import watchdog.*; import java.rmi.*;

public class Server {

 public Main() {

  composition BookCopyComponent =
<BookCopy_SubjectConnector<BookCopyComponent> {
   {BookCopy_SubjectConnector:changeOp <- {
        BookCopy::void borrowIt(),
        BookCopy::void returnIt()
     }
   }
  }

  composition ClientComponent = ...

  composition BookManagerComponent = ...

  BookManager myBookMan = new BookManagerComponent();
  Naming.rebind("rmi://ip:port/BookManager", myBookMan);

  BookCopy bc = new BookCopyComponent();
  myBookMan.buy(bc);
  //notification registers myBookMan
  //as observer of bc.
 }
}
```

Since the predicate expression of all specializing methods in BookManager_ObserverConnector invariantly evaluates to true, the feature identifier "notification" is activated by default for the myBookMan instance. The code of the

notification feature will constructively cast the bc reference to interface Subject and the myBookMan instance will be registered as Observer of bc.

7.3 Context-sensitive late binding

Figure 3 showed an execution trace where three separate client programs concurrently customize the behavior of the library application by selecting different variant features of the reusable-subject-observer collaboration. In figure 8 we zoom into a specific client program to study in further detail how global combination consistency is achieved.

Propagation of composition policies. First of all, context-sensitive late binding is applied to whole collective object behavior instead of single objects. Within a given client-specific view, all messages are thus consistently dispatched to the same combination of features. This is due to the propagation of the composition policy with the message flow.

```
import digital-library.*;
import java.rmi.*;

public class ClientProgram2 {

 public Main() {
  BookManager myBookMan = (BookManager)
     Naming.lookup("rmi://ip:port/BookManager");

  Client cl2 = myBookMan.createClient();

  cl2@watchdog;
  //client-specific selection of the
  //watchdog feature and, implicitly,
  //the notification feature.

  BookCopy bc1 = cl2.selectBook("...");

  cl2.borrowBook(bc1);
  //notification will register cl2
  //as observer with bc1, because bc1 was already
  //borrowed by cl1

}
```

Figure 8: The code of Figure 3 stripped down to a specific client program.

Consistency in the presence of callbacks. The second rule for ensuring global combination consistency, namely that the composition policy should stick when an object passes itself as argument in a message, is demonstrated when looking at the implementation of the reusable-subject-observer collaboration. Here, when an Observer object registers itself with an Subject object, a new reference is created at the Subject object. It is important to understand that the feature identifiers, which are in the composition policy, must be transparently annotated to the newly created reference, hence the expression that composition policies should 'stick'. This is required because the subject potentially calls the observer back and we want to preserve global combination consistency in the

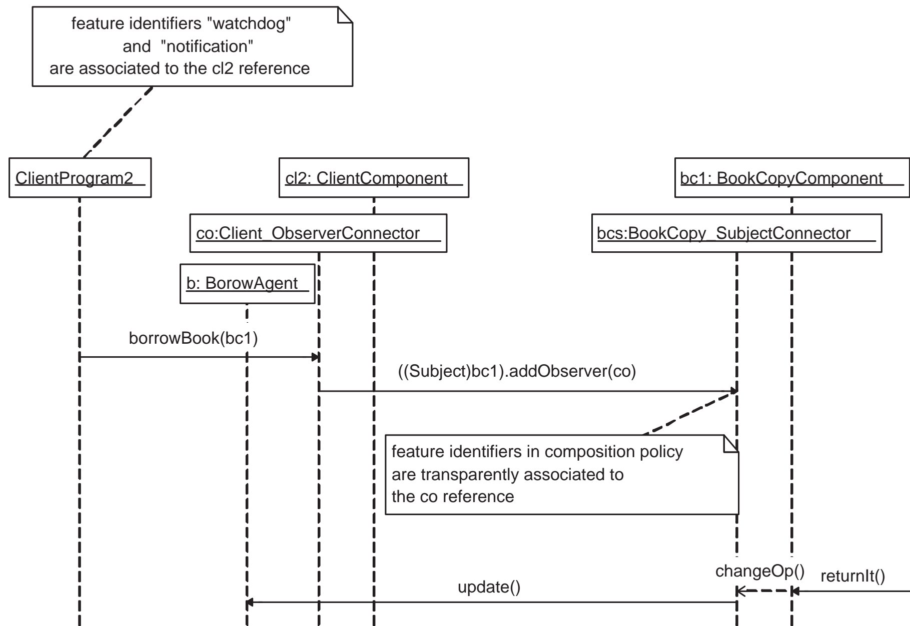

Figure 9: System-wide consistency in the combination of collaborations

presence of such callbacks: when a subject notifies the observer, we want to execute at least the same combination of features that were activated when the observer registered itself with the subject.

The transparent annotation of the composition policy is straightforward to realize based on the propagation of the composition policy with the message flow. This is graphically presented in more detail in figure 9. The figure presents the message flow that is executed when the ClientProgram class invokes the borrowBook() operation on the cl2 reference of type Client. Notice that at that execution point, there is already an association between the cl2 reference and the "watchdog" feature (and, implicitly, the "notification" feature[9]). Since the requested bookcopy bc1 has already been borrowed by another client, Client_ObserverConnector registers Client cl2 as Observer with Bookcopy bc1. It is important to understand that the reference to cl2[10], that bc1[11] receives, is transparently annotated with the "watchdog" and "notification" feature identifiers. This ensures system-wide consistency in the presence of callbacks: a subsequent invocation of the update() operation on cl2 will be transparently redirected to the BorrowAgent instance. This is consistent with the fact that the client cl2 initially has activated the "watchdog" feature.

[9] Remember that watchdog depends on notification

[10] More precisely, this is cl2's Observer instance that is named co in figure 9

[11] More precisely, this is bc1's Subject instance that is named bcs in figure 9

8 Discussion

In this section we evaluate the Lasagne object model. First the properties of the LOM are summarized, then some related work is described.

8.1 Evaluation

To synthesize and evaluate the Lasagne object model, we revisit the eight requirements that were enumerated in section 3.

Context-sensitive late binding is supported because messages are dispatched based on their attached feature identifiers and feature identifiers can be attached by the usage context[12].

The principle of *encapsulation* is respected in two senses: (*i*) message-passing clients have no access to the specialization interface of objects because this issue is governed by composition patterns. And (*ii*) class-based structure is not exposed to message-passing clients because they only refer to interfaces and feature identifiers, which also belong to interfaces.

Local behavioral consistency is supported by propagation of the composition policy.

The LOM can be easily implemented in *distributed execution environments* because the essential concept of the LOM – feature identifiers – are serializable and thus can easily be propagated with distributed message flow.

Receiver-controlled late binding is supported by means of a simple variant of predicate dispatch[14].

Large-scale features are implemented as collaborations between multiple components and by labelling these with the same feature identifier.

Global combination consistency is also supported by propagation of the composition policy. To preserve consistency in the presence of callbacks, the composition policy should "stick" to passed self references.

Finally, the *genericity* of the LOM has been illustrated by explaining that it subsumes single delegation and qualified message passing without exposing their problems.

8.2 Related work

This section discusses how the Lasagne object model and the Lasagne programming language relates to other work.

8.2.1 Enhanced object models

To our knowledge their does not exist in the literature an object model that satisfies all eight requirements as the LOM does. Of course, no research project is an island. So there is certainly overlap of existing work. We describe those approaches that have strong overlaps with one or more properties of the LOM and we describe these overlaps only.

[12] and by the operating environment, but this paper has not gone into details about this.

The *composition filters* (CF) model[2] is an aspect-oriented extension to the conventional object model. It is based on manipulating incoming and outgoing messages of objects. The CF model adopts declarative specifications. These describe what should be done, rather than how. Declarative specification thus supports a powerful language for programming aspectual behavior. A newer version of the CF model includes the so-called superimposition specification[6]. The superimposition specification describes the places in the program where aspect behavior is to be added. This new CF model supports thus composition of large-scale features. As opposed to the LOM, the CF model does not provide coordination mechanisms to preserve global combination consistency in the presence of context-sensitive selection of features. Since the underlying architecture of the CF model is very generic, however, the necessary coordination mechanisms can be added; incoming messages can be delegated to a meta-object that implements the LOM.

The *Mixin Methods* approach[35, 36] enhances the traditional class-based object model with controlled delegation. Here, a parent object statically specifies the set of child objects that are allowed to delegate to this parent. Furthermore special-purpose methods encapsulate the construction and navigation through the delegation hierarchy of the object. The Mixin Methods has also been incorporated in a language for programming distributed applications, named Pico[25], enabling remote sender-initiated late binding. Therefore, with respect to intra-object dispatch, the object model of the Mixin Methods approach shares many of the benefits of the LOM. However, it does not provide support for predicate dispatch.

Split objects [4] provides a per viewpoint representation of objects. A split object is defined as a collection of pieces. Pieces are organized within a delegation hierarchy and forming an object. As opposed to split objects, pieces have no identity. Pieces can be added to and removed from split objects as needed. Messages are sent on a per viewpoint basis by giving an identifier of a piece. It is also possible to send messages without any viewpoint. Thus, with respect to intra-object dispatch and except support for predicate dispatch, the split objects model is very similar to the LOM.

The *Rondo object* model[26, 27] presents a sophisticated approach to dynamic and context-sensitive object evolution without name collisions. The main contribution of Rondo is that it allows to realize visibility control, inheritance, and dynamic object modification orthogonally to each other. The Rondo object model shares some similarities with the LOM because Rondo introduces the notion of scope identifiers that drives method dispatch. Scope identifiers are however used for resolving name collisions and supporting flexible visibility control, and not for context-sensitive selection of methods.

Unless has been stated otherwise, the LOM distinguishes itself from the above object models by its feasibility to be implemented in distributed execution environments, its support for large-scale features and the context-sensitive selection thereof, and its support for preserving global combination consistency.

8.2.2 Advanced object-oriented language constructs

This section relates the LOM and the Lasagne programming language to research that supports feature-based dispatch by stretching the conventional object model to its corners and, thus, that designs new language constructs on top of the conventional object model.

To our knowledge, there is only one approach in this category, namely the Delegation Layers (DL) approach[31]. The DL approach combines delegation with the notion of dependent types[30, 12]. Inspired by the Mixin Layers approach[33], DL organizes a collaboration as a set of classes that are nested within an outer class. Two collaborations are dynamically composed by composing their respective outer classes by means of delegation. As a result, the nested classes of these collaborations are implicitly composed by means of delegation as well. The DL approach brings delegation thus at the level of collaborations. The notion of dependent types is used to enforce consistency in the composition of collaborations. Family polymorphism ensures that the nested classes are virtual, meaning that the behavior of the instances of the nested classes is dependent on the identity and type of their aggregating outer object. The identity of the outer object ensures consistency in the combination of collaborations; since the self parameters of the different nested class instances are dependent on the same outer object, they are always interpreted in harmony to the same set of collaborations.

Combining delegation and dependent types thus yields a language construct that supports dynamic composition of collaborations while preserving global combination consistency. A major remark in this respect is that DL does not provide any consistency guarantees in the presence of callbacks. The dispatch of a callback in DL is only driven by the client-specific view that has (indirectly) initiated that callback.

With respect to composition of collaborations (i.e. composition of their components), the Lasagne programming language is less dynamic than DL. Composition patterns in Lasagne can be determined at runtime, but only once during bootstrap. We have chosen this approach deliberately because in service-oriented architectures, it is not at all desired that clients can add features to remote servers at free will. Client-specific selection of features is enough.

Another striking difference between DL and Lasagne is that in DL a whole collaboration implementation is lexically nested in one class whereas in Lasagne a collaboration is implemented as different components that may be dispersed across different processes or hosts. By applying lexical nesting, DL allows to localize the code of a collaboration in one place which eases understanding and maintenance. We have not applied lexical nesting, however, because it stands in the way of independently developed components: lexical nesting does not allow to define collaborations whose components are developed independently from each other. In Lasagne, it is possible to establish a collaboration between two independently developed components based on an earlier agreed abstract collaboration. It is also possible to deploy these components at different nodes of a distributed system. In short, it is not clear whether the DL approach can

be implemented in a distributed environment.

A fourth difference is that DL provides weak object identity at the outer object level. When a client program wants to select a particular combination of collaborations it has to refer to the corresponding outer object. This may lead to complex maintenance of references and duplicate state problems at the outer object level. The application programmer must deal with these problems manually by using factory objects that canonicalize[17] the outer objects. In Lasagne, annotation of feature identifiers is the means to express context-sensitive selection of features and as such there is no need to switch references.

Finally, a fifth important difference is that DL does not provide support for receiver-controlled late binding.

8.2.3 Aspect-oriented programming languages

In this section we describe how the LOM (and the Lasagne programming language) relates to other work in the area of aspect-oriented programming languages (AOPL's).

The pioneer AOPL's such as Aspect/J[20] and Hyper/J[38] allow to modularize crosscutting large-scale features. Feature dispatch is not supported in these languages however. Aspect/J does provide a special pointcut construction called `cflow` that allows the programmer to define aspects that must only be activated within specific control flows. However, this mechanism is completely different from feature-based dispatch: feature-based dispatch is concerned with a dynamic selection of which aspects to execute, not just whether to invoke a given aspect or not[13]. In other words, as stated by [13], Aspect/J lacks support for aspectual polymorphism.

Newer AOPL's such as Caesar[28] and Object Teams[16] allow to dynamically select aspects at runtime. Similar to Lasagne, these languages makes a distinction between static composition and dynamic selection of collaborations. The major difference with the Lasagne programming language is that these languages provide no support for remote selection of aspects. Nor is it possible to distribute an aspect implementation across multiple hosts.

8.2.4 Component-oriented programming

It seems that our definition of components and objects differs considerably from the original definition of components by Szyperski[37]. Szyperski considers a component as a unit of modularity that is composed of multiple classes. Thus, the composition relationship between components and objects seems completely inverted in the arena of component-oriented programming: the runtime instantiation of a single component consists of multiple objects, whereas the Lasagne object model states it exactly the other way around.

This is however just appearance. In reality, the Lasagne object model conforms with Szyperski's definition. This is because collaboration-based design provides the necessary concepts for building large-scale components. Thus, we

regard collaborations as composite components that encapsulate a set of finer-grained components.

8.2.5 Meta-programming

A final area of related research is meta-programming. Meta-programs are programs that manipulate (control) other programs by means of a meta-interface[21].

In an object-oriented setting, the meta-interface consists of objects, the so-called meta-objects. A meta-object represents the structural and computational aspects of an associated application object. The meta-object protocol[19] (MOP) is the documented interface to these metaobjects. To clearly distinguish the "normal" application objects from the metaobjects, the former are often called baselevel objects[32].

The Lasagne programming language offers a very simple metaprogramming facility in the form of multi-signature methods that manipulate messages as first-class entities. However, modern meta-object protocols are much more powerful than that. The Lasagne object model can actually be implemented using a MOP. As a disadvantage, meta-programming is often too complex for an average application programmer to comprehend. The Lasagne programming language is easier to understand.

Another difference with the Lasagne programming language is that a meta-level program will generally apply its behavior to all messages of the underlying base level, whereas in Lasagne, multi-signature methods will only apply its behavior to a well-defined subset of messages. This last point makes Lasagne more suitable for implementing functional features that are tightly related to the functionality of a particular application.

9 Conclusion

Feature-oriented decomposition and composition capabilities are a very useful complement to object-oriented programming because it makes software more adaptable to evolving requirements. Yet, feature dispatch is not supported by the conventional object model.

In this paper we have presented the Lasagne object model that extends the conventional object model with feature-based dispatch by incorporating the notion of feature identifiers. Message-senders can influence method dispatch by specifying the feature identifiers to be activated, whereas message-receivers can impose additional features upon clients by means of predicates. Sender-initiated late binding is especially interesting in distributed systems because clients can customize the functionality of remote services by selecting feature identifiers instead of switching remote references by means of complicated middleware. This dynamic feature selection also preserves system-wide consistency in the combination of features.

We have presented the requirements for an extended object model that supports feature dispatch. Second, we have presented a programming language

that implements the LOM and we have illustrated the overall concept with a feature-based application. Third, we have included a formal basis of the LOM and, fourth, we have discussed how the LOM is a solid basis for recent programming language research; we have referred to and compared with various research-level programming languages that already direct towards stronger support for feature-based composition.

References

[1] A. M. A. Rashid and J. Araújo. Modularisation and composition of aspectual requirements. In *2nd International Conference on Aspect-Oriented Software Development*, pages 11–20. ACM, 2003.

[2] M. Akşit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer-Verlag, 1994.

[3] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 158–167. IEEE Computer Society, May 2004.

[4] D. Bardou and C. Dony. Split objects: a disciplined use of delegation within objects. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 122–137. ACM Press, 1996.

[5] D. Batory, R. Cardone, and Y. Smaragdakis. Object-oriented frameworks and product-lines. In *Software Product Lines*, volume 576 of *The Kluwer International Series in Engineering and Computer Science*, pages 227–247. Kluwer Academic Publishers, 1999.

[6] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[7] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41(5):257–273, 1999.

[8] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, SIGPLAN Notices 25(10), pages 303–311. ACM Press, 1990.

[9] M. Büchi and W. Weck. Generic wrappers. In *ECOOP 2000, 14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer, 2000.

[10] B. Carré and J.-M. Geib. The point of view notion for multiple inheritance. In *OOPSLA/ECOOP '90 Proceedings*, SIGPLAN Notices 25(10), pages 312–321. ACM Press, 1990.

[11] S. Danforth and I. R. Forman. Reflections on metaclass programming in SOM. In *OOPSLA*, pages 440–452, 1994.

[12] E. Ernst. Family polymorphism. In *ECOOP 2001—Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001.

[13] E. Ernst and D. H. Lorenz. Aspects and polymorphism in AspectJ. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 150–157. ACM Press, 2003.

[14] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211. Springer, 1998.

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

[16] S. Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2002.

[17] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 36–56. Springer-Verlag, 1993.

[18] B. N. Jørgensen. LasagneJ website. http://www.lasagnej.org.

[19] G. Kiczales, J. des Riviers, and D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP2001 – Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[21] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 481–490. ACM Press, 1997.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. L., and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[23] G. Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP '99—Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 1999.

[24] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA'86*, SIGPLAN Notices 21(11), pages 214–223. ACM Press, 1986.

[25] W. D. Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2004.

[26] M. Mezini. Dynamic object evolution without name collisions. In *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 190–219. Springer, 1997.

[27] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publishers, 1998.

[28] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100. ACM Press, 2003.

[29] S. Microsystems. Java technology. http://java.sun.com/, 1994-2005.

[30] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003—Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer-Verlag, 2003.

[31] K. Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002—Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, 2002.

[32] B. Robben. *Language Technology and Metalevel Architectures for Distributed Objects*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 1999.

[33] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 1998.

[34] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86 Conference Proceedings*, ACM SIGPLAN Notices, 21(11), pages 38–45. ACM Press, 1986.

[35] P. Steyaert, W. Codenie, T. D'Hondt, K. D. Hondt, C. Lucas, and M. V. Limberghen. Nested mixin-methods in agora. In *Proceedings ECOOP '93*, volume 707 of *Lecture Notes in Computer Science*, pages 197–219. Springer-Verlag, 1993.

[36] P. Steyaert and W. D. Meuter. A marriage of class-and object-based inheritance without unwanted children. In *ECOOP'95-Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 127–144, 1995.

[37] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, 1998.

[38] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press / ACM Press, 1999.

[39] E. Truyen. Lasagne implementation on top of Correlate. http://www.cs.kuleuven.ac.be/~eddy/lasagne.html.

[40] E. Truyen. *Dynamic and Context-Sensitive Composition in Distributed Systems*. PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, 2004. pages 207-224.

[41] P. Wegner. Dimensions of object-based language design. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 168–182, 1987.

A Formal specification

In this appendix we formally specify a subset of the computational viewpoint of the LOM. In particular we specify the LOM as a method dispatch mechanism for object-oriented languages. It does not include a treatment of the constructive cast operator, multi-signature methods, and type-annotated references. The formalization is intended for illustrative and explanatory purposes rather than for verification purposes.
Let $OBJECT$ be the set of objects, L be the set of all selectors, MSG the set of all messages, FI the set of all feature identifiers, $CONTEXT$ the set of all contextual properties and $reject$ the non-existent element.

A.1 Message structure

The notion of a message $msg \in MSG$ is defined as

$msg = (s, n, l, cp, context)$ where
(i) s is the sender object of the message, n is the receiver object and $l \in n.I$ is the selector of the message.

(ii) $cp \in 2^{FI}$ is the composition policy of the message. 2^{FI} is the set of all possible subsets of FI, i.e. a composition policy is a set of feature identifiers $\{fi_1, \ldots, fi_n\} \in 2^{FI}$.
(iii) Similarly, $context \in 2^{CONTEXT}$ represents the contextual properties of the message.

A.2 Object

An object $o \in OBJECT$ is defined as

$o = ((CO, \leq_p), I, M, S, R, value, featureMap, self, mc, interceptor)$ where
(i) CO is the set of component instances of o. $\leq_p$ denotes all precedence relationships between the elements of CO. CO is totally ordered with regard to $\leq_p$. Thus, CO can be represented as $(< co_k, co_{k-1}, \ldots, co_0 >, \leq_p)$ which is a totally ordered set where $\forall\ i : 1 \ldots k,\ co_{i-1} \leq_p co_i$ means that co_i has precedence over co_{i-1}.
(ii) I is the interface of o and is defined as $\bigcup_{co \in CO} co.I$, i.e. I equals the union of selectors of all component instances in CO.
(iii) Likewise, $M = \bigcup_{co \in CO} co.M$.
(iv) S is the state space formed by the instance variables of the component instances of o, i.e. $S = \Pi_{co \in CO} co.S$.
(v) $featureMap : FI \rightarrow (CO \vee reject)$ is a mapping that associates the component instances of o with their unique feature identifier. Thus, $\forall\ co \in CO$: the feature identifier of co is $featureMap^{-1}(co)$. If $featureMap$ maps a given $fi \in FI$ to $reject$ then there exists no component instance in o that is bound to that feature identifier fi.
(vi) The set of references R of o is defined as a subset of $(FI \times \bigcup_{co \in CO} co.R)$. Specifically, $o.R =$

$\{\ (fi, t) \mid fi \in FI\ \wedge\ featureMap(fi) \neq reject\ \wedge\ t \in featureMap(fi).R)\ \}$.
(vii) The associated reference mapping $value$ is then defined as

$o.value : (o.R \cup \{self\}) \rightarrow (OBJECT \cup (MSG \rightarrow CO))$

where

- $\forall\ r = (fi, t) \in o.R : o.value(r) =$
  $featureMap(fi).value(t) \in OBJECT$
- $o.value(self)$ is a function of type $MSG \rightarrow CO$

$(viii)$ $self$ is the pseudo-reference that a method of a component instance of o has to use to send a message to the whole object o. $self$ is defined as a reference whose value is a function of type $MSG \rightarrow CO$ that in its turn is computed by another function $match : (2^{CO}, \leq_p) \rightarrow (MSG \rightarrow CO)$.

The value of $self$ is thus defined as:

$o.value(self) = \ match(CO)$

$match$ is defined as follows. Let $cos \in (2^{CO}, \leq_p)$ and $msg \in MSG$. Then $match(cos)$ is a function of type $MSG \rightarrow CO$ such that $match(cos)(msg)$ gives as result the component instance $c \in cos$ with highest precedence whose feature identifier is listed in the composition policy $msg.cp$ and that has a matching method for selector $msg.l$.

Formally, $match : (2^{CO}, \leq_p) \rightarrow (MSG \rightarrow CO)$ is defined by the following logical expression. Let $< co_h, \ldots, co_0 > \in (2^{CO}, \leq_p)$, $msg \in MSG$. Then,

$$\begin{array}{c} match(< co_h, \ldots, co_0 >)(msg) \ = \ c \in \{co_h, \ldots, co_0\} \\ \Longleftrightarrow \\ match_cnd(msg, c) \\ \wedge \\ (\forall co_j \in \{co_h, \ldots, co_0\} : match_cnd(msg, co_j) \implies co_j \leq_p c) \end{array}$$

where $match_cnd : MSG \rightarrow CO \rightarrow \{true, false\}$ is defined as:

$$\begin{array}{c} match_cnd(msg = (..., ..., l, cp, ...), co) = true \\ \Longleftrightarrow \\ featureMap^{-1}(co) \in cp \ \wedge \ co.P(l) \neq reject \end{array}$$

To sum up, the value of $self$ is a function that is returned by calling $match$ where we have passed CO as first parameter. As a result, the value of $self$ defines a runtime evaluation of itself. This runtime evaluation function is shared by all component instances of o.
(ix) $mc : MSG \rightarrow M$ is the method combiner of o. For messages received from other objects and calls to $self$, mc passes the message to the runtime evaluation function that computes the value of the $self$ parameter. Thus:

$mc(msg = (..., self, l, cp, context)) =$
$(o.value(self)(msg)).P(l)$

(x) $interceptor : MSG \rightarrow MSG$ represents the interceptor of object o. $interceptor$ is a function that updates the composition policy of messages, i.e.:

$interceptor(s, n, l, cp, context) = (s, n, l, cp', context)$ where $cp' \in 2^{FI}$

A.3 Component instances

Given an object $o = ((CO, \leq_p)$, I, M, S, R, $value$, $featureMap$, $self$, mc, $interceptor)$. Then each component instance $co_i \in CO$ is defined as

$co_i = (I, M, S, P, R, value, this, inner)$ where
(i) M, S, P, R and $this$ are the same as defined by the conventional object model.

(ii) The interface I of the component instance co_i is explicitly specified as provided services in the component program metadata of co_i. Non-public methods are not part of the interface. Thus, I is defined as

$I = \{selector(m) \mid m \in M \wedge (m.visibility = public)\}$ where $selector : M \rightarrow L$ is a function that returns the selector by which method m can be invoked.
(iii) The *value* mapping of co_i is defined as

$co_i.value : (co_i.R \cup \{this, inner\}) \rightarrow (OBJECT \cup CO \cup (MSG \rightarrow CO))$

where

- $\forall r \in co_i.R, co_i.value(r)$ is the object $q \in OBJECT$ to which r refers.
- $co_i.value(this)$ is $co_i \in CO$ itself.
- $co_i.value(inner)$ is a function of type $MSG \rightarrow CO$

(iv) $co_i.inner$ is the pseudo-reference that a method of co_i has to use to forward a received message to a signature-compatible method of another component instance. $co_i.inner$ is, similar to $self$, defined as a reference whose value is a function of type $MSG \rightarrow CO$ that is again computed by function $match : (2^{CO}, \leq_p) \rightarrow (MSG \rightarrow CO)$. Specifically, the value of $co_i.inner$ is defined as

$co_i.value(co_i.inner) \;=\; match(< co_{i-1}, \ldots, co_0 >)$

The value of $co_i.inner$ is a function that is returned by calling match where we have passed as first parameter the component instances over which co_i has precedence (thus $\{co_r \in CO \mid co_r <_p co_i\}$).
(v) The method combiner mc of the enclosing object o must also map *inner* calls to the appropriate method. Specifically, for a call to *inner* from a method of component instance co_i, the message is passed to the runtime evaluation function that computes the effective value of co_i.'s *inner* parameter. Thus:

$mc(msg = (co_i, co_i.inner, l, cp, context)) =$
$(co_i.value(co_i.inner)(msg)).P(l)$

Calls to *this* from a method of component instance co_i are directly passed to mapping P of co_i itself. Thus, mc does not dispatch calls to *this*.
(vi) Finally, to understand the propagation of composition policies and contextual properties, we must specify the behavior of component instances. Let $cp \in 2^{FI}$ and $context \in 2^{CONTEXT}$. Then, the behavior B of component instance co_i is defined as

$B(co_i, cp, context) = \{b_m \;: S \rightarrow (messages(cp, context), S') \mid m \in co_i.M\}$

where $messages : 2^{FI} \rightarrow 2^{CONTEXT} \rightarrow 2^{MSG}$ is defined as:

$messages(cp, context) \;=$

$\{$
$\quad (co_i, n, l, cp, context') \in MSG \mid$

$n \in (\ co_i.value(co_i.R \cup \{co_i.this, co_i.inner\})\ \cup\ o.value(self)\ ),$
$l \in L,$
$context' \in CONTEXT$

}

Thus, the composition policy is propagated unchanged with the message flow whereas contextual properties may be updated at any execution point.

A.4 Predicate dispatch

The formalization, presented till thus far, contains the core concepts of the LOM. Support for predicate dispatch is specified as an incremental extension of these core concepts.

First, we must extend the method structure. Suppose an object $o = ((CO, \leq_p), I, M, S, R, value, featureMap, self, mc, interceptor)$, then $\forall\ co \in CO$, the methods $co.M$ of co are associated with a predicate $pred : [(MSG, co.S) \rightarrow \{true, false\}$ or no predicate ($pred = nil$).

Second, the checking of predicates can be modelled by means of a redefined *match* function. The redefined function is defined as follows. Let $cos \in 2^{CO}$ and $msg \in MSG$. Then $match(cos)(msg)$ returns the component instance $co \in cos$ with highest precedence that has a matching method for selector $msg.l$, and

- the predicate expression associated to that method evaluates to true, or
- there is no predicate expression associated to that method and the feature identifier of co is listed in the composition policy $msg.cp$.

The above redefinition can be formally specified by redefining the boolean function $match_cnd : MSG \rightarrow CO \rightarrow \{true, false\}$ as follows. Let $msg \in MSG$ and $co \in CO$. Then:

$$\begin{gathered} match_cnd(msg = (..., ..., l, cp, ...), co) = true \\ \Longleftrightarrow \\ co.P(l) \neq reject \\ \wedge \\ (\quad (\ co.P(l).pred = nil\ \wedge\ featureMap^{-1}(co) \in cp\ ) \\ \vee \\ (\ co.P(l).pred \neq nil\ \wedge\ co.P(l).pred(msg, co.S) = true\ )\quad) \end{gathered}$$