Superimposition: A Component Adaptation Technique

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Jan.Bosch@ide.hk-r.se
www: http://www.ide.hk-r.se/~bosch

Abstract
Several authors have identified that the only feasible way to increase productivity in software construction is to reuse existing software. To achieve this, component-based software development is one of the more promising approaches. However, traditional research in component-oriented programming often assumes that components are reused "as-is". Practitioners have found that "as-is" reuse seldomly occurs and that reusable components generally need to be adapted to match the system requirements. Existing component object models provide only limited support for component adaptation, i.e. white-box techniques such as copy-paste and inheritance and black-box approaches such as aggregation and wrapping. These techniques suffer from problems related to reusability, efficiency, implementation overhead or the *self* problem. To address these problems, this paper proposes *superimposition*, a novel black-box adaptation technique that allows one to impose predefined, but configurable types of functionality on a reusable component. Three categories of typical adaptation types are discussed, related to the component interface, component composition and component monitoring. Superimposition and the types of component adaptation are exemplified by several examples.

1 Introduction

Component-oriented programming is receiving increasing amounts of interest in the software engineering community. The aim is to create a collection of reusable components that can be used for component-based application development. Application development then becomes the selection, adaptation and composition of components rather than implementing the application from scratch. The concept of component-oriented programming almost seems analogous to object-oriented programming. The notion of a component refers to an module that contains both code and data and presents an interface that can be invoked by other components.

The naive view of component reuse is that the component can just be plugged into an application and reused as is. However, many researchers, e.g. [Hölzle 93, Samentinger 97, Yellin & Strom 94], have identified that "as-is" reuse is very unlikely to occur and that in the majority of the cases, a reused component has to be adapted in some way to match the application's requirements. Adapting a component can be achieved in several ways, but traditional techniques can be categorised into *white-box*, e.g. *inheritance* and *copy-paste*, and *black-box*, e.g. *wrapping*, adaptation. The white-box adaptation techniques generally require understanding of the internals of the reused component, whereas the black-box adaptation techniques ideally only require knowledge about the component's interface.

In this paper, we argue that the aforementioned techniques are insufficient to deal with all required types of adaptation without experiencing, potentially considerable, problems. Examples of these problems are the lack of reusability of components, since they cannot be adapted, and the adaptation specification itself. In addition, the software engineer may spend considerable effort on understanding the component before being able to adapt it. Also, non-transparent adaptation techniques may lead to the *self* problem [Lieberman 86] and, finally, the adapting a component may require considerable amounts of code with extremely simple behaviour, e.g. forwarding a message.

To address these problems, we introduce the notion of *superimposition*, a technique that enables the software engineer to impose predefined, but configurable, types of functionality on a component's functionality. Examples of superimposing behaviour are interface adaptation as in the *Adapter* design pattern [Gamma et al. 94], client specific interfaces and implicit invocation. Superimposition allows the software engineer to adapt a component using a number of predefined adaptation behaviours that can be configured for the specific component. Since one may identify new types of adaptation behaviour, the software engineer can define new adaptation types. Finally, the software engineer may compose multiple adaptation behaviour types for a single component.

The notion of superimposition has been implemented in the layered object model (**LayOM**), an extensible component object language model. **LayOM** consists, next to conventional object model elements as *nested objects* (instance variables) and *methods*, of *states*, *categories* and *layers*. The layers encapsulate the basic object and all messages sent to or from the object are intercepted by the layers. Through the use of these layers, **LayOM** provides several types of superimposing behaviour that can be used to adapt components. The advantage of layers over traditional wrappers is that layers are transparent and provide reuse and customisability of adaptation behaviour.

The contribution of this paper, we believe, is that the requirements that a component adaptation technique should fulfil are identified and the problems associated with traditional component adaptation techniques are presented. In addition, a new, complementing technique, i.e. superimposition, is proposed. Finally, a collection of useful types of adapting behaviour that we identified is proposed that improves the reusability of components and can directly be supported in the language model through superimposition.

The remainder of this paper is organised as follows. Next, the running example used in this paper, i.e. dialysis systems, is introduced. In section 2, conventional component adaptation techniques are described in more detail and evaluated for the requirements. Section 3 discusses the notion of superimposing adaptation behaviour on components from a conceptual perspective and a number of typical types of adaptation are described. Section 4 presents the layered object model and presents two example layer types. Three examples of superimposing adaptation behaviour, i.e. interface adaptation, delegation of requests and observer notification, are discussed in section 5. These examples are taken from the dialysis system example discussed below. In section 6, our results are compared to related work and the paper is concluded in section 7.

1.1 Example: Dialysis System

The example that is used throughout the paper is taken from one of our industrial projects, i.e. dialysis systems. From a software architecture perspective, a dialysis system consists of a hierarchical collection of devices, each with its own controlling strategy and alarm handler. The software consists of a graphical user interface module, a control system module and a protective system module. The GUI module provides an interface to the medical and technical staff at the hospital where the dialysis system is used, whereas the control system contains the logic for the actual dialysis process, including communication with the hardware sensors and actuators. The protective system is responsible for the patient's safety and continuously monitors the actions by the control system. Whenever a potentially dangerous situation occurs, the protective system intercepts and stops the dialysis process.

The control system can be divided into two major parts, i.e. the hemo-dialysis fluid part and the blood part. The two systems meet in a membrane, where water and waste products travel from the blood into the dialysis fluid. Despite its perceived simplicity, a dialysis system is a very complex embedded system with a multitude of sensors, for e.g. temperature, flow, heparin, concentration, etc., and actuators, e.g. pumps, heaters, valves, etc. An abstraction of the control system part is presented graphically in figure 1.

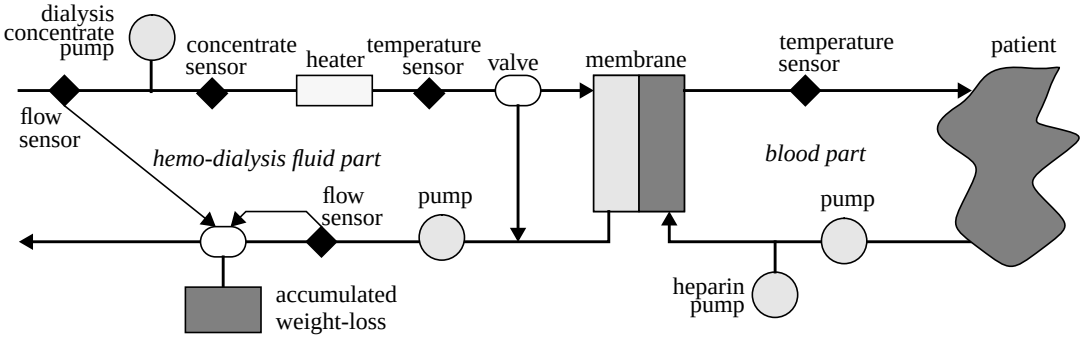

Figure 1. Overview of the dialysis system

The industrial project aimed at providing an architecture design for the dialysis system, incorporating its specific requirements with respect to changeability and demonstrability. Since dialysis systems need to be certified by independent certification institutes, the system should be designed such that it is easy to demonstrate the system's correct, or at least safe, behaviour. The architecture design provided the top-level structure for the components that make up the system behaviour. In the project, several of the components could be reused from earlier system versions. However, most reused components needed to be adapted in some way. In this paper, we present the various types of adaptation that we identified in this project as well as other projects and examples of component adaptation are presented.

2 Component Adaptation Techniques

Component-based software engineering intends to construct applications by putting together reusable components. This perspective on software construction, although becoming more widespread during recent years, was already mentioned during the late sixties [Samentinger 97]. Components, however, never became one of the leading software development paradigms, despite the early optimism. This, we believe, is due to a rather naive view on software construction based on components as exists within the software engineering community. The naive view imagines that one selects a set of components that deliver parts of the application requirements and then put these components together by connecting inputs to outputs. However, research in software reuse has shown that 'as-is' reuse occurs extremely little and that components generally need to be changed in some way to match the application architecture or the other components. The process of changing the component for use in a particular application is often referred to as *component adaptation*.

Over time, research in software engineering and programming languages has developed a number of techniques for adapting components. These component adaptation techniques can be categorised into white-box and black-box adaptation techniques. White-box techniques require the software engineer to adapt a reused component either by changing its internal specification or by overriding and excluding parts of the internal specification. Black-box techniques reuse the component as it is, but adapt at the interface of the component. Black-box adaptation only requires the software engineer to understand the interface of the component, not the internals.

Obviously, the above division is rather extreme and in practice, e.g. inheritance often requires the understanding of only part of the internal functionality whereas wrapping may require more understanding of the component than just its interface specification. In the remainder of this section, we first present the requirements that a component adaptation technique should fulfil. Subsequently, the conventional techniques are presented and evaluated with respect to the requirements.

2.1 Requirements for Component Adaptation Techniques

Before discussing conventional component adaptation techniques, the requirements that a component adaptation technique has to fulfil in general is specified. These requirements provide a framework that can be used to evaluate conventional component adaptation techniques, but also to provide an insight in the required functionality of novel adaptation types.

- **Transparent**: The adaptation of the component should be as *transparent* as possible. Transparent, in this context, indicates that both the user of the adapted component and the component itself are unaware of the adaptation in between them. In addition, aspects of the component that do not need to be adapted should be accessible without explicit effort of the adaptation. Wrapping a component, for instance, requires the wrapper to forward all requests to the component, including those that need not be adapted.
- **Black-box**: The software engineer always has to develop some mental model of the functionality of a component before the component can be reused. This model should, however, be kept as small and simple as possible. This requires that the adaptation technique needs no knowledge of the internal structure of the component, but is limited to the interface of the component.
- **Composable**: The adaptation technique should be easily composable with the component for which it is applied, i.e. no redefinition of the component should be required. Secondly, the adapted component should be as composable with other components as it was without the adaptation. Finally, the adaptation should be composable with other adaptations. In the situation where default adaptation types are available, it may be that the component needs to be adapted using multiple types of adaptation. This requires the various adaptations to be composable.
- **Configurable**: Adaptation generally consists of a generic and a specific part. For example, the adaptation type *changing operation names* has a generic part, i.e. replacing the selector in the message with another name and a specific part, i.e. which selectors should be replaced with what names. For the adaptation technique to be useful and reusable, the technique has to provide sufficient configurability of the specific part.
- **Reusable**: A problem of traditional adaptation techniques is that both the generic and the specific part are not reusable. Since the two aspects are so heavily intertwined, the generic part cannot be separated from the specific part and, consequently, the software engineer is forced to reimplement the adaptation over and over again. A new technique should provide reusability of the adaptation type and particular instances of the adaptation type, i.e. both the generic and the specific part.

2.2 Adaptation Techniques

When using a conventional object-oriented language, the software engineer has three component adaptation techniques that can be used to modify a reused component, i.e. *copy-paste*, *inheritance* and *wrapping*. In the next sections, each technique is described and subsequently evaluated with respect to the identified requirements.

2.2.1 Copy-Paste

When an existing component provides some similarity with a component needed by the software engineer, the most effective approach may be to just copy the code of that part of the component that is suitable to be reused in the component under development. After copying the code, the software engineer will often make changes to it to make it fit the context of the new component and additional functionality will be defined or copied from other sources. [Samentinger 97] refers to this technique as *code scavenging*.

Although the copy-paste technique provides some reuse, it obviously has many disadvantages, among others the fact that multiple copies of the reused code are existing and that the software engineer has to intimately understand the reused code. However, from our discussions with professional software engineers and students, we were surprised to see how often this technique is applied, especially when time pressure or other factors may force for a “quick-and-dirty” approach.

With respect to the aforementioned requirements, the evaluation of the copy-paste technique can be summarised as follows:

- **Transparent**: Since the reused code and new code are merged into a new component, there are no problems associated with transparency. Both the reused component as the client of the adapted component notice no difference between the reused code and the code for adaptation.
- **Black-box**: Since there is no encapsulation boundary between the component code and the adaptation code, the black-box requirement is not fulfilled at all.
- **Composable**: Due to the merging of code, composability of adaptation functionality with the reused component is very low. In cases where one would want to compose several types of adaptation behaviour, the software engineer has to merge all code manually.
- **Configurable**: Adapting a component through copy-paste does not represent the adaptation behaviour as a first-class entity, thus no configurability is available.
- **Reusable**: Since the adaptation behaviour has no first-class representation and is intertwined with the code of the reused component, no reuse of either the component or the adaptation behaviour is possible, except through the same copy-paste behaviour.

2.2.2 Inheritance

A second technique for white-box adaptation and reuse is provided by inheritance. Inheritance as provided by, e.g. Smalltalk-80 and C++, makes the state and behaviour of the reused component available to the reusing component. Depending on the language model, all internal aspects or only part of the aspects become available to the reusing component. For instance, in Smalltalk-80 [Goldberg & Robson 89] all methods and instance variables defined in the superclass become available to the subclass, where in C++, it depends on use of the *private* and *protected* keywords what methods and instance variables become available to the subclass. Inheritance provides the important advantage that the code remains to exist in one location. However, one of the main disadvantages of inheritance is that the software engineer generally must have detailed understanding of the internal functionality of a superclass when overriding superclass methods and when defining new behaviour using behaviour defined in the superclass.

With respect to the requirements, the evaluation of inheritance can be summarised as follows:

- **Transparent**: Since the subclass implicitly forwards messages to the superclass, inheritance in transparent. The reused component, i.e. the superclass, as well as client objects using instances of the introduced sub-class notice no difference.

- **Black-box**: Whether inheritance is black-box, depends primarily on its implementation in the language model. In Smalltalk-80, all instance variables and methods become available to the subclass, leaving no boundary between the reused component and the adaptation code. Other language models, e.g. C++ and Java, allow for private instance variables and methods, thus being able to separate the component from the adaptation behaviour.
- **Composable**: Although the adaptation behaviour is specified in a subclass and thus separated from the component, it is still difficult to compose the adaptation behaviour with another component or to associate multiple adaptation behaviours with a class. Even though some languages would allow for simple changing of the name of the superclass for an adaptation type, the problem is still that the generic and specific parts of the adaptation are merged and are difficult to separate.
- **Configurable**: As mentioned, although the adaptation behaviour is represented as a first-class entity, i.e. a subclass, inheritance provides no means to configure the specific part of the adaptation behaviour.
- **Reusable**: Despite the fact that inheritance facilitates the representation of the adaptation type as a class, it may prove difficult to reuse it since the name of the adapted component is hard-wired in the class and because it cannot be configured.

2.2.3 Wrapping

Wrapping declares one or more components as part of an encapsulating component, i.e. the wrapper, but this component only has functionality for forwarding, with minor changes, requests from clients to the wrapped components. There is no clear boundary between wrapping and aggregation, but wrapping is used to adapt the behaviour of the enclosed component whereas aggregation is used to compose new functionality out of existing components providing relevant functionality. An important disadvantage of wrapping is that it may result in considerable implementation overhead since the complete interface of the wrapped component needs to be handled by the wrapper, including those interface elements that need not be adapted. Also, others, e.g. [Hölzle 93], have identified that wrapping may lead to excessive amounts of adaptation code and serious performance reductions.

The evaluation of wrapping with respect to the requirements is the following:

- **Transparent**: Since the wrapper completely encapsulates the adapted component, clients of the component can not send messages to the component directly but always need to pass the wrapper. This requires the wrapper to handle all messages that could possibly be sent to the component, including those messages that do not need to be adapted.
- **Black-box**: Since the component is accessed by the wrapper as any client, i.e. through the interface, the wrapping technique is black-box. The wrapper has no way to access or depend upon the internals of the adapted component.
- **Composable**: Wrapping is, different from the other conventional techniques, composable. A wrapper and its wrapped component form again a component that can be wrapped by another wrapper. This process can be repeated recursively. However, since the wrappers are not transparent, each wrapper needs to implement the complete interface of its wrapped component in order to be used in all cases where the unadapted component could be used.
- **Configurable**: Although the adaptation behaviour is represented as a first-class entity, i.e. a wrapper, generally no means to configure the specific part of the adaptation behaviour are available. For instance, when the wrapper needs to change the name of an operation at the adapted component, it is generally not possible to configure the wrapper with the new operation name since this has to be hard coded in the wrapper.
- **Reusable**: The wrapper can be reused in those cases where exactly the same adaptation behaviour is required. However, since wrappers cannot be configured, every difference from the original case makes as-is reuse impossible and requires either the wrapper to be edited or the combination of wrapper and component to be adapted by a new wrapper.

2.3 Evaluating Conventional Techniques

In table 1, an overview of the conventional adaptation techniques is presented that indicates how well each technique fulfils the specified requirements. From the table, one can see that some problems are dealt with well wrapping but not so well by the white-box techniques, i.e. copy-paste and inheritance, and visa versa

Requirement	Copy-Paste	Inheritance	Wrapping
transparent	+	+	-
black-box	--	-	+
composable	-	-	+
configurable	-	-	-
reusable	-	-	+/-

Table 1. Conventional adaptation techniques versus the identified problems and requirements

The copy-paste technique, as well as inheritance, is transparent since the reused and adaptation behaviour are merged in a single entity. However, on the other requirements, the white-box adaptation techniques do not score so well. Wrapping is not transparent, since it encapsulates the adapted component. Wrapping is black-box by definition and wrapping is composable since a wrapped component can again be wrapped by another wrapper adapting different aspects of the original component. Configurability and reusability are not well supported by traditional techniques since no distinction between generic behaviour and component-specific behaviour is made. Due to this, it is not possible to separate the generic aspects and apply them for a different component.

Concluding, none of the conventional component adaptation techniques fulfils the requirements that are required for effective component-based software engineering. Therefore, one may deduce alternative approaches are required. In this paper we propose superimposition as a technique to component adaptation.

3 Component Adaptation through Superimposition

As was identified in the previous section, traditional component adaptation techniques do not fulfil all the requirements one would put on them. Certain types of functionality need to be integrated with the component's behaviour that are orthogonal to its structural parts and may affect e.g. multiple methods. The software engineer needs to *superimpose* certain behaviour on a component in such a way that the complete functionality of the component is affected. The notion of superimposition in computing systems has been identified before, but not in object-oriented or component-based systems. For example, [Bouge & Francez 88] define and use superimposition in the context of CSP. They define the superimposition *R* of *P* over *Q* as the additional superimposed control *P* over the basic algorithm *Q*. Analogously, we define object superimposition *S* of *B* over *O* as the additional overriding behaviour *B* over the behaviour of component *O*. Different from inheritance, a single unit of superimposed behaviour can change several aspects of the basic object's behaviour.

Superimposition as a concept is a very suitable technique for adapting components in a component-based system. The principle underlying superimposition is that a component and the functionality adapting the component are two separate entities on the one hand and need to be very tightly integrated on the other hand. Often, both the component and the adaptation behaviour are reusable as independent entities, whereas the combination, i.e. the adapted component is too specific for the current application to be reusable in future applications. Constructing an application using reusable components primarily is the activity of selecting and configuring a set of interacting components, where part of the component configuration is the adaptation of the component to fit the requirements of the current application.

Based on the above observation, we have identified that component-based software engineering, in addition to a set of reusable components, requires a set of reusable component adaptation types. These adaptation types should be configurable and composable with each other to allow for complex component adaptations. In section 3.2, several types of component adaptation are identified and presented.

Since component adaptation types are composed with the component and other adaptation types, this composition needs to be transparent in its nature. In other words, the component and clients of the component should be unaware of the presence of adaptation entities. In addition, adaptation entities should be unaware of the presence of other adapta-

tion entities that are active for the same component. In figure 2, superimposition is graphically represented. A basic component is shown that is adapted using two adaptation types. The adapted component is encapsulated by the adaptations, but clients of the component nor the component itself note any of these differences.

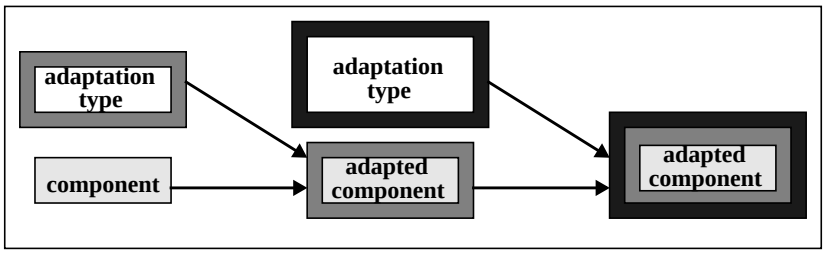

Figure 2. Component adaptation through superimposition

3.1 Defining Superimposition

To define the informally introduced notion of superimposition in more detail, in this section a more precise object model specification is developed. The formalisation is intended for illustrative and explanatory purposes rather than for verification purposes.

A *object* o is defined as $o = (I,M,S,P)$ where *I* indicates the interface of the object, *M* the set of methods, *S* the state space formed by the instance variables and *P* the mapping from the interface to the methods. The interface of the object is the set of message selectors that the object can respond to. For a basic object, $I = \{m \in M | selector(m)\}$ where $selector(m)$ is a function that returns the selector using which method *m* can be invoked. The mapping *P* is defined as $P{:}I \rightarrow M \vee reject$, i.e. each interface element is either mapped to a method in the method set or rejected. Also, we define the behaviour *B* of an object as $B = \{m \in M | b_m \Rightarrow S \rightarrow S\}$. The behaviour b_m of a method *m* is defined as the state change of the object caused by an execution of *m*. Finally, the notion of a message is defined as $e = (n, r, l)$, where *n* is the sender of the message, *r* is the receiver and *l* is the selector. We do not incorporate the message arguments in the model for reasons of simplicity, neither do we incorporate state changes at other objects due to messages sent by a method executed at o.

An object can have a set of superimposing entities *G* associated with it; $G = \{g_1, \ldots, g_p\}, p \geq 0$. A superimposing entity *g* can be composed with an object *o*. The composition $o' = g \otimes o$ leads to a new object o' that is an adaptation of *o* since aspects of it may be changed, i.e. $g \otimes o = \langle I',M',S',P' \rangle$. However, the result is an object that, for a client of the object, is indistinguishable from a basic object. A superimposed object o can be defined as $g_1 \otimes \ldots \otimes g_p \otimes o, p \geq 1$.

3.2 Component Adaptation Types

During our work on component adaptation, we have identified three typical categories of component adaptation, i.e. component interface changes, component composition and component monitoring. In the sections below, each of these categories is discussed in more detail.

3.2.1 Changes to Component Interface

A typical situation in component-based system construction is when a component in principle could be reused in the system at hand, but its interface does not match the interface expected by the system. Typical examples are that operations have the wrong names or that the interface contains irrelevant operations. In such situations, the interface of the component, in order for the component to be reused, needs to be adapted to match the expected interface. Below, some typical examples of component interface adaptation are presented.

- **Changing operation names**: Perhaps the most typical problem when reusing a component is that the names of some of the operations provided by the component do not match the expected interface. This problem has been identified by many software engineers and even an *Adapter* design pattern has been defined [Gamma et al. 94].

Although the pattern conceptually exactly does what one can expect, its implementation suffers from several problems as we identified in [Bosch 97]. In any case, changing the operation names of a component is a type of adaptation that has to be provided by a adaptation technique.

This adaptation type is defined as a superimposing entity type $g^{ad}(I^{ad}, P^{ad})$ where $I^{ad} = \{i_1^{ad}, ..., i_q^{ad}\}, q \geq 1$ and $P^{ad}:I^{ad} \rightarrow M$. The composition of an object o with an instance of g^{ad} has the semantics $g^{ad}(I^{ad}, P^{ad}) \otimes o = (I \cup I^{ad}, M, S, P \cup P^{ad})$. Thus, the interface of the object is extended with an set of additional interface elements and a mapping function for the new interface elements. The original operation names are not removed from the interface.

- **Restricting parts of the interface**: A second change to the component interface that a component may require is the exclusion of a part of the interface. In the reusing context, a part of the interface may not be relevant or even counter-productive, e.g. for typing reasons, if it would be accessible by clients of the component. Adaptation of the component should then restrict access to the relevant operations.

  Interface restriction can be formally defined as a superimposing entity type $g^{res}(I^{res})$ where $I^{res} = \{i_1^{res}, ..., i_q^{res}\}, q \geq 1$. The composition of an object o with an instance of g^{res} has the semantics $g^{res}(I^{res}) \otimes o = (I–I^{res}, M, S, P – \{p:i \rightarrow m \wedge i \in I^{res}\})$. Thus, the set of specified interface elements is removed from the interface of the object and the mappings for the removed interface elements are excluded from the mapping function *P.*

- **Client and state-based restriction**: In systems where a component is used by clients of various types, the component may need to act in several roles, see e.g. [Reenskaug et al. 95]. This requires the component to present a tailored interface to each client type, i.e. each client has only access to that part of the interface that it requires. This we refer to as client-based interface restriction. In addition, parts of the interface of the component may be accessible or restricted based on the *state* of the component. For instance, an empty buffer component is unable to provide an element to a client requesting a “get” operation. This we refer to as state-based interface restriction. Both types of interface restriction are important types of component adaptation.

  Client-based interface restriction can be formally defined as a superimposing entity type $g^{cl}(c, I^{cl})$ where *c* refers to the client for which the interface is to be restricted and $I^{cl} = \{i_1^{cl}, ..., i_q^{cl}\}, q \geq 1$. The composition of an object o with an instance of g^{cl} has the semantics

$$g^{cl}(c, I^{cl}) \otimes o = \begin{cases} (I–I^{cl}, M, S, P – \{\forall i \in I^{cl} \mid p:i \rightarrow m\}) \text{ when } e = (n, r, l) \wedge n = c \\ (I, M, S, P) \text{ otherwise} \end{cases}$$

  Thus, for messages $e \in E$, where E is the set of all messages, sent by client object *c,* the set of specified interface elements is removed from the interface of the object and the mappings for the removed interface elements are excluded from the mapping function *P.* For messages sent by other objects, the semantics of the object remain unchanged.

  State-based interface restriction is defined analogous to client-based interface restriction. Only the conditions based on which interface elements are made inaccessible is now based on the object state, rather than on the client sending the message. Thus, a a superimposing entity type $g^{st}(s, I^{st})$ where *s* is defined as $s:S \rightarrow Boolean$, i.e. a function mapping the object state to a boolean value and $I^{st} = \{i_1^{st}, ..., i_q^{st}\}, q \geq 1$ is the set of interface elements that is excluded when the state is true. The composition of an object o with an instance of g^{st} has the semantics

$$g^{st}(s, I^{st}) \otimes o = \begin{cases} (I–I^{st}, M, S, P – \{\forall i \in I^{st} \mid p:i \rightarrow m\}) \text{ when } s = \textit{true} \\ (I, M, S, P) \text{ otherwise} \end{cases}$$

3.2.2 Component Composition

One of the early phases in system construction often is of a top-level, system design in which the components of the system and their interactions are defined. Once it is known what components are needed in the system, the available collections of reusable components are searched to identify potentially reusable components. In such situations, it may occur that the functionality that a component should provide in the system design cannot be fulfilled by a single com-

ponent. However, a combination of two or more components is able provide the required functionality. In such cases, the components have to composed such that the resulting structure seems a single component from the system's perspective. Below, three types of component adaptation relevant for component composition are discussed.

- **Delegation of requests**: The easiest way for a component to providing required services not available within the component itself is to delegate a request for such a service to another component that is able to provide the requested service. To achieve this, the component needs to be extended with behaviour that delegates certain requests to other components.

  Message delegation can be achieved using a superimposing entity type $g^{del}(o^d, I^{del})$ where o^d indicates the object to which messages should be delegated and I^{del} defines the set of interface elements for which messages should be delegated to the object. The semantics of the composition of g^{del} and an object o is:

$$g^{del}(o^{del}, I^{del}) \otimes o = \left(I \cup I^{del}, M, S, P \cup \left\{ p \in P_{O^d} \middle| p{:}i \rightarrow m_{O^d} \wedge i \in I^{del} \right\} \right)$$

- **Component composition**: In cases where two components need to be more structurally integrated, the two components can be aggregated in a encapsulating component. In [Gamma et al. 94] the *Facade* pattern is defined for this purpose. However, the encapsulating component needs to delegate requests to the contained components such that the requirements of the system are fulfilled. The traditional approach would be to define a large collection of small methods at the encapsulating component that forwards the messages to the correct encapsulated component. This approach, obviously, leads to considerable implementation overhead for the software engineer. In addition, the reusability of the solution is very limited. The underlying cause for these problems is that aggregation is not transparent.

  When superimposition is available as a composition technique, the solution is to define a type of superimposing entity that allows one to compose two or more objects without the aforementioned disadvantages. The defined entity is $g^{comp}(P^{expl})$ where P^{expl} defines an explicit mapping function that can be defined by the software engineer when instantiating a g^{comp}. The composition of g^{comp} with two or more objects has the following semantics

$$g^{comp}(P^{expl}) \otimes (o_1, \ldots, o_q) = (I_{o_1} \cup \ldots \cup I_{o_q}, M_{o_1} \cup \ldots \cup M_{o_q}, S_{o_1} \cup \ldots \cup S_{o_q}, P)$$

  where $P = \left\{ \left(P_{o_1} \cup \ldots \cup P_{o_q} - \left\{ \forall p \in P_{o_1} \cup \ldots \cup P_{o_q}, p{:}i \rightarrow m | \exists p' \in P^{expl}, p{:}i \rightarrow m' \right\} \right) \cup P^{expl} \right\}$

  Informally, the above specification states that the superimposing entity will forward all messages transparently to the nested component that defines a corresponding method. However, in case of name conflicts or otherwise, the software engineer can define an explicit mapping for interface elements. The explicit mapping function overrides the implicit definition.

- **Acquaintance selection and binding**: No component is an island, i.e. virtually all components require other components, acquaintances, to provide them with services in order to be able to deliver the functionality needed by the system. However, since the designers of reusable components are unable to make all but minimal assumptions about the context in which the component will operate, the binding of the acquaintances required by the component is often performed in an ad-hoc manner, e.g. when the component is instantiated. As we identified in [Bosch 96c], the traditional acquaintance binding omits several important aspects. Component adaptation should allow for flexible, expressive specification of the way acquaintances are selected and bound.

  The above description of acquaintance handling requires that, for every object, the set of accessible objects is available. In concrete computer systems, this service is often provided by a broker architecture. However, for the formal specification of the required semantics of acquaintance handling we assume that a set `Objects` is available and directly accessible to the superimposing entity. A superimposing entity $g^{acq}(a, cnd)$ is defined for acquaintance handling in which *a* is the acquaintance identifier and *cnd* defines the condition that has to be fulfilled by a potential acquaintance object in order to be selectable. For reasons of simplicity, we have left the outward mapping function *Q* out of the object specification presented earlier. *Q* defines how a message $e_o = (n, r, l)$ send by object

o, i.e. $o = n$, is bound to objects in the context of *o*. The composition of g^{acq} and an object *o* have the following semantics: $g^{acq}(a, cnd) \otimes o = (I, M, S, P, Q')$ where

$$Q' = \begin{cases} q{:}r \rightarrow o_r \text{ where } \{o \in Objects | cnd(o)\} = \begin{cases} \varnothing \Rightarrow (nil \rightarrow o_r) \\ \{o_1, \ldots\} \Rightarrow (o_1 \rightarrow o_r) \end{cases} & \text{if } r = a \\ q{:}r \rightarrow o_r \text{ where } \{o \in Objects | name(o) = name(r)\} & \text{if } r \neq a \end{cases}$$

Informally, the adaptation type will bind a message send by *o* to a receiver *a* to the first object in the set of objects that fulfil the specified condition. If no object fulfils the condition, the message is bound to the *nil* object. If the receiver of the message is not *a*, the normal, name-based binding will take place.

3.2.3 Component Monitoring

The previously discussed categories of component adaptation are concerned with changing the behaviour of the reused component, e.g. its interface. This category is, as the name implies, primarily concerned with the monitoring of the component so that other components are notified or invoked when certain events at the monitored component occur. Below, we discuss three examples of monitoring that can be superimposed on reusable components, i.e. implicit invocation, observer notification and state monitoring. The latter types are specialisations of the first one.

- **Implicit invocation**: This type actually is the general adaptation type for component monitoring. The concept of implicit invocation is concerned with notifying relevant components, either directly by message sending or indirectly through event generation, whenever certain conditions or actions take place at the monitored component.

  The superimposing type corresponding to the above description is $g^{ii}(o_m, l_m, I^{ii})$ where o_m indicates the object that is to be implicitly invoked, and I^{ii} the set of interface elements that should cause an implicit invocation to o_m. The semantics of the composition of g^{ii} and an object *o* is.

  $$g^{ii}(o_m, l_m, I^{ii}) \otimes o = (I, M, S, P \cup \{\forall i \in I^{ii} | p{:}i \rightarrow m \Rightarrow p{:}i \rightarrow (m, o_m \cdot l_m)\})$$

  Informally, the mapping function is extended with a mapping from the interface elements in I^{ii} to $o_m \cdot l_m$. Thus, messages calling elements in I^{ii} lead to two invocations, i.e. the original method invocation and the implicit notification invocation.

- **Observer notification**: In [Gamma et al. 94], one of the presented design patterns is the *Observer* pattern. This pattern explains how the relation between some object and a set of objects depending on the state of that object should be implemented. This pattern was originally introduced in the Smalltalk-80 system as *model-view-controller* (MVC). Although the pattern is extremely useful, it presumes that the software engineer knows, when defining an object, that it will be observed by other objects. In component-based system construction, the reused components sometimes need to be observed, but generally the component is not prepared for this. Therefore, the observer pattern functionality needs to be superimposed on the component so it can be used as an observed component. The adaptation behaviour is both responsible for the administration and the notification of dependent objects.

  The formal specification only defines the notification behaviour; the administrative behaviour is left as an exercise for the reader. The observer adaptation type is a specialisation of the implicit notification type. The adaptation type is $g^{obs}(D, I^{obs})$ where D defines the set of dependents objects and I^{obs} the set of interface elements that should cause an notification to the dependents. The semantics of the composition of g^{obs} and an object *o* is.

  $$g^{obs}(D, I^{obs}) \otimes o = (I, M, S, P \cup \{\forall i \in I^{ii} | p{:}i \rightarrow m \Rightarrow p{:}i \rightarrow (m \cup \{\forall (d \in D | (d \cdot changed))\})\})$$

  Informally, the mapping function is extended with a mapping from the interface elements in I^{ii} to all dependent objects. Compliant to the observer pattern definition, the changed method is invoked at the dependent objects.

- **State monitoring**: In some cases, dependent components do not want to be notified for every state change in the observed component, but only when the component state exceeds certain boundaries. The conventional observer pattern behaviour is not prepared for this, but using superimposition it is well feasible to implement this behaviour. This allows the software engineer reusing the component to specify in what state regions the dependent components need to be specified.

The monitoring adaptation type is defined as $g^{mon}(o_m, l_m, S^{mon})$ where o_m is the monitoring object and S^{mon} the monitored state. The composition of g^{mon} and an object *o* is:

$$g^{mon}(o_m, l_m, S^{mon}) \otimes o = (I, M, S, P \cup \{\forall m \in M, b_m{:}S \rightarrow S^{mon} \,|\, p{:}i \rightarrow (m, o_m \cdot l_m)\})$$

Informally, invocations to all methods that cause state changes in that part of the object state that the monitoring object is interested in will lead to invocation of the monitoring object, in addition to the normal method execution.

3.3 Composing Adaptation Types

Nine types of component adaptation have been introduced, organised in three categories. However, all types were presented as individual component adapters and the relation to other adapters was not discussed. This is because the result of each composition of a superimposing adaptation type and an object again is an object, although with altered semantics, i.e. $g \otimes o = o'$. Thus, when a component is adapted by several adapters, each adapter will assume to be adapting an object and has no knowledge of or reference to the other adapters. For example, assume an object composed with three adapters $g_3 \otimes g_2 \otimes g_1 \otimes o$. This structure can recursively be reduced to any object, i.e. $g_3 \otimes g_2 \otimes g_1 \otimes o = g_3 \otimes g_2 \otimes o_1 = g_3 \otimes o_2 = o_3$.

As an example, we present an object $o = (\{a, b\}, \{m_a, m_b\}, \{x, y\}, \{p{:}a \rightarrow m_a, p{:}b \rightarrow m_b\})$ and two adaptation types, i.e. $g^{del}(o^{del}, \{c, d\})$ and $g^{ad}(\{e, f\}, \{p{:}e \rightarrow m_a, p{:}f \rightarrow m_d\})$. The composition of these entities results in to an object with the following semantics:

$$g^{ad} \otimes g^{del} \otimes o = (\{a, \ldots, f\}, \{m_a, m_b\}, \{x, y\}, \{p{:}a \rightarrow m_a, p{:}b \rightarrow m_b, p{:}c \rightarrow o^{del} \cdot c, p{:}d \rightarrow o^{del} \cdot d, p{:}e \rightarrow m_a, p{:}f \rightarrow m_d\})$$

However, when composing an object and one or more adaptation types, conflicts between the different entities may occur. For instance, in the previous example, a restricting adapter could have been used that would hide some of the interface elements provided by the delegating adapter. Since each adapter superimposes on the encapsulated object *and* all adapters in between the adapter and the object, the effect of encapsulated adapters can be reduced or removed.

3.4 Evaluating Superimposition

Having criticized the conventional adaptation techniques for not fulfilling the requirements, it is of course only fair to also evaluate the superimposition technique with respect to these requirements.

- **Transparent**: The adaptation types that can be superimposed on components are fully transparent. The identity of the component is preserved and only those aspects of the component behaviour that need to be adapted are actually affected by the adaptation type.
- **Black-box**: Superimposing adaptation types are fully black-box in that they do not depend on the actual implementation of the component nor is the implementation of the adaptation type visible to the component, its clients or other adaptation types.
- **Composable**: As was discussed in the previous section, the adaptation types can freely be composed with each other due to the fact that adapted components cannot be distinguished from other components.
- **Configurable**: Adaptation types consist of a generic and a specific part. The specific part can be defined for each instantiation of the adaptation types, making it configurable.
- **Reusable**: The generic part of adaptation types is reused for all instantiations and, as such, provides the superimposition technique the highest possible level of reusability.

4 Layered Object Model

The layered object model (**LayOM**) is our research language model that has successfully been applied to several problems associated with the traditional object-oriented paradigm. The layers that are part of the model provide an implementation of superimposition through the use of message interception. However, **LayOM** is only one implementation

approach to superimposition and alternative approaches to implementing superimposition can be developed. The remainder of this section is organised as follows. In the next section, the basic features of the layered object model are described. In section 4.2, some example layer types are presented. Section 4.3 is concerned with defining new types of component adaptation.

4.1 LayOM

The layered object model is an extended object model, i.e. it defines in addition to the traditional object model components, additional components such as layers, states and categories. In figure 3, an example LayOM object is presented. The layers encapsulate the object, so that messages send to or by the object have to pass the layers. Each layer, when it intercepts a message, converts the message into a passive message object and evaluates the contents to determine the appropriate course of action. Layers can be used for various types of functionality. Layer classes have, among others, been defined for the representation of relations between classes and between objects [Bosch 96a], design patterns [Bosch 97] and acquaintance selection and binding [Bosch 96c].

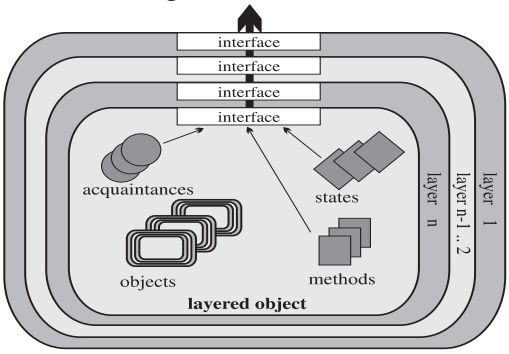

Figure 3. The layered object model

A **LayOM** object contains, as any object model, instance variables and methods. The semantics of these components is very similar to the conventional object model. The only difference is that instance variables, as these are normal objects, can have encapsulating layers adding functionality to the instance variable.

A state in **LayOM** is an abstraction of the internal state of the object. In **LayOM**, the internal state of an object is referred to as the concrete state. Based on the object's concrete state, the software engineer can define an externally visible abstraction of the concrete state, referred to as the abstract state of an object. The abstract object state is generally simpler in both the number of dimensions, as well as in the domains of the state dimensions.

An acquaintance category is an expression that defines a set of objects that are treated similarly by the object. This set of objects treated as equivalent by the object we denote as acquaintances. A category describes the discriminating characteristics of a subset of the external objects that should be treated equally by the class. The behavioural layer types use acquaintance categories to determine whether the sender of a message is a member of the category. If the sender is a member, the message is subject to the semantics of the specification of the behavioural layer type instance.

A layer, as mentioned, encapsulates the object and intercepts messages. It can perform all kinds of behaviour, either in response to a message or pro-actively. Layers are primarily used to represent relations between objects, for the representation of design patterns and for acquaintance handling. In **LayOM**, relations have been classified into structural relations, behavioural relations and application-domain relations. Structural relation types define the structure of a class and provide reuse. This relation type can be used to extend the functionality of a class. The second type of relations are the behavioural relations that are used to relate an object to its clients. The functionality of the class is used by client objects and the class can define a behavioural relation with each client (or client category). Behavioural relations restrict the behaviour of the class. For instance, some methods might be restricted to certain clients or in specific situations. The third type of relations are application domain relations. Many domains have, next to reusable application domain classes, also application domain relation types that can be reused. For instance, the `controls` relation type is a very important type of relation in the domain of process control. In the following section, structural relation layer types and acquaintance handling will be discussed in more detail.

Next to being an extended object model, the layered object model also is an extensible object model, i.e. the object model can be extended by the software engineer with new components. **LayOM** can, for example, be extended with new layer types, but also with new structural components, such as events. The notion of extensibility, which is a core

feature of the object-oriented paradigm, has been applied to the object model itself. Object model extensibility may seem useful in theory, but in order to apply it in practice it requires extensibility of the translator or compiler associated with the language. In the case of **LayOM**, classes and applications are translated into C++. The generated classes can be combined with existing, hand-written C++ code to form an executable. The implementation of the **LayOM** and its support for language extensibility are discussed in section 4.3.

4.2 Representing relations and acquaintance handling

Superimposition of component adaptation behaviour is provided by the layers in the layered object model. The available layer types provide reusable and configurable types of component adaptation. However, in order for the examples in section 5 to make sense, two example layer types are presented below. The next section discusses the representation of inter-object relations, in particular partial inheritance, whereas acquaintance selection and binding is discussed in section 4.2.2.

4.2.1 Structural relations

Structural relation types define the structure of an application. A class uses the structural relations to extend its behaviour and the class can be seen as the client, i.e. the class that obtains functionality provided by other classes. Generally, three types of structural relations are used in object-oriented systems development: inheritance, delegation and part-of. These types of relation all provide some form of reuse. The inherited, delegated or part object provides behaviour that is reused by, respectively, the inheriting, delegating or whole object. Therefore, next to referring to these relation types as structural, we can also define them as reuse relations.

Orthogonal to the discussed relation types one can recognise two additional dimensions of describing the extended behaviour of an object, i.e. conditionality, and partiality. Conditionality indicates that the reusing object limits the reuse to only occur when it is in certain states. Partially indicates that the reusing object reuses only part of the reused object.

Due to space constraints, it is not possible to describe the syntax and semantics of all structural relation types. Instead, one layer of type `Inherits` is described in detail. For an extensive description of the semantics of the other structural relation types we refer to [Bosch 96a].

An example class `TemperatureSensor` contains a partial inheritance layer with the following configuration:

```
pin: Inherits(Sensor, *, (calibrate));
```

The semantics of the inheritance layer are that a part of the interface of the inherited class is reused or excluded. The name of this layer is `pin` and its type is `Inherits`. The layer type accepts three arguments. The first argument is the name of the class that is inherited from; `Sensor` in this case. The second argument is a '*' or a list of interface elements and indicates the interface elements that are to be inherited. The '*' in this example indicates that all interface elements are inherited. The third argument defines the excluded interface elements and can either contain a '*' or a list of interface elements. In this example, the list only consists of one element: `calibrate`. The semantics of layer `pin` is that class `TemperatureSensor` inherits the complete interface of class `Sensor`, except for `calibrate`.

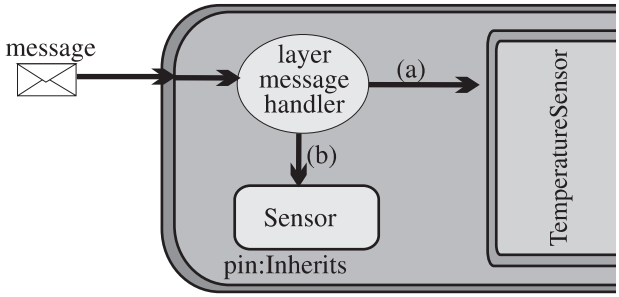

Figure 4. The `Inherits` layer.

In figure 4, the implementation of this semantics is illustrated. The inheritance layer is the second layer of class `TemperatureSensor`. There is the most outer layer, shown around layer `pin` and an inner layer, shown around `TemperatureSensor`. An inheritance layer creates an instance of the inherited superclass, in this case an instance of class `Sensor`. The layer contains a message handler, that, for each received message, determines whether the message is

passed on inwards or outwards or that it is redirected to the instance of class `Sensor`. In the figure, an incoming message is shown. The message is reified and handed to the message handler. The message handler will read the selector field of the message and compare it with the (partially) inherited interface of class `Sensor`. If the selector is part of the set of interface elements, the message is redirected to the instance of class `Sensor` (situation (b) in figure 3). If the selector does not match with the interface of class `Sensor`, it is not redirected but forwarded to the next layer (situation (a) in figure 3).

4.2.2 Acquaintance layers

Almost any object, in the course of its operation, communicates with other objects for achieving its own goals. We refer to the objects that an object needs to communicate with as acquaintances. An acquaintance may provide some services to the object, it might request services from the object or it may both request and provide services. In object-oriented systems, the amount of work required from the software engineer to connect the various objects should be as little as possible. The object itself should contain the specifications on how to connect to its acquaintances and what requirements a potential acquaintance should fulfil. On the other hand, when an object is used in an unanticipated context, the software engineer requires expressive and modular language constructs to connect an object to the other objects.

As discussed in [Bosch 96b], one can identify a number of problems associated with the way traditional object-oriented languages deal with acquaintance handling, among others, related to reusability and expressiveness. Within the context of the layered object model, a layer of type `Acquaintance` has been defined. The syntax of the layer type is as follows:

<layer-id> : Acquaintance([<obj-name> | <category>] is-bound [permanent | per-call | from <selector> until <selector>]
for [one | all | <n>] objects from [inside | <n> contexts | global]);

An instance of this layer type is bound based on the name of a called object or based on the name of a category. The actual object that is communicated with may have been bound permanently on object instantiation, bound for every call or during a transaction starting with a `<selector>` and ending with the same or another `<selector>`. When selecting the appropriate object to bind the acquaintance to, the search may be inside the object itself, in `<n>` contexts of the object or globally. The notion of contexts is important in our underlying system view where the objects in the system are organised hierarchically. The first context of the object consists of those objects that are located in the immediate vicinity of the object, i.e. in the same subsystem, etc.

To illustrate the use of the acquaintance layer type, we discuss the dialysis system class as an example. A dialysis system object has a number of acquaintances, i.e. a patient object, nurse object and a manufacturer object. Since the system is constantly used by different patients and operated by different nurses, no permanent bindings for these acquaintances can be specified. Instead, these acquaintances are bound when the situation is appropriate. Below, part of the **LayOM** specification for the mobile phone class is shown.

```
class DialysisSystem
    layers
        nurse : Acquaintance(nurse is-bound per-call for one object from 1 contexts);
        patient : Acquaintance(patient is-bound from connectNeedle until dialysisFinished for one object from global);
        mf : Acquaintance(manufacturer is-bound permanent for one object from global);
        ...
    categories
        patient begin acq.subClassOf(Person) and acq.kidneyPatient(); end;
        nurse begin acq.hasDegree("HealthCare") or dialysisUnit.hasEmployee(acq); end;
        manufacturer begin self.manufacturer() = acq; end;
    ...
end;
```

The specification only contains the layer and category definitions for the aforementioned acquaintances of the dialysis system object. The categories describe the discriminating characteristics of the acquaintance objects, whereas the layers describe how to bind actual objects to each acquaintance. The object name `acq` used in the category specifications is a pseudo variable used to refer to the acquaintance being defined. Note that one could have defined the `DialysisSystem` class without the `Acquaintance` layers. That allows one to add the layer specifications to individual instances of the `DialysisSystem` class, thus creating unique instances. For a more detailed description of acquaintance handling we refer to [Bosch 96c].

4.3 Defining New Component Adaptation Types

The layered object model is an extended but also an *extensible* object model. Extensibility, in this context, refers to the ability of the language model to be extended with new components, such as new layer (component adaptation) types. Since traditional compiler technology is unsuitable to implement extensible languages, an alternative approach had to be defined. The implementation of the layered object model is based on the notion of *delegating compiler objects* (DCOs) [Bosch 96b]. A DCO is an object that compiles a part of the syntax of the input language. It consists of one or more lexers, one or more parsers and a parse graph. The nodes in the parse graph have the ability to generate code for themselves. In case of the **LayOM** class compiler, it consists of a class DCO, method DCO, state DCO, category DCO and a DCO for each layer type. Each DCO definition results in a class and a DCO object can instantiate another DCO and delegate control to it. The delegated DCO will perform its functionality and return control to the delegating DCO when it is finished. The **LayOM** compiler DCOs generate C++ output code[1]. **LayOM** code is either a class or an application. A **LayOM** class is compiled into a C++ class and a **LayOM** application is compiled into a C++ main program. The generated C++ class can be incorporated in any C++ function and, subsequently, into an executable program.

An advantage of using the DCO approach is that it supports extensibility of the language very well. When the software engineer wants to add, for example, a new type of component adaptation, all that is required is a DCO defining the syntax and code generation information of that particular concept. The new DCO is added to the set of DCOs in the existing compiler and it can be used immediately. In some cases it is necessary to make some minor modifications in the DCOs that should instantiate the new DCO, but these cases are rare.

It is, however, not necessary nor intended that every software engineer implements a private set of component adaptation types. In most cases, the provided types of component adaptation will be sufficient to impose the required adaptations. In slightly more complicated cases, multiple adaptation types can combined. Occasionally, it really is necessary to define a new component adaptation type and in those cases a company or project can assign a software engineer to implement a DCO that provides the required adaptation behaviour. The developed DCO tools, however, considerably simplify this task.

5 Adapting Components for the Dialysis System

To illustrate the use of component adaptation types, the dialysis system will be used as an example. As mentioned, the first phase in the project was to design a software architecture for the system that would define the main components and their interactions. Since earlier versions of the dialysis system existed, the intention was to use the components from earlier dialysis systems where ever possible. As one may expect, no component could be reused ‘as-is’, i.e. each reused component needed to be adapted up to some extent. The company was forced to implement some components from scratch, even though similar components for older system versions existed. For the actual system, the company used a conventional object-oriented language, i.e. C++, to build their dialysis system. One of the activities within the academic part of the project was to investigate how a more high-level language model could address the problems related to component adaptation.

In this section, three selected examples from the dialysis system are introduced that are addressed by superimposition, but caused problems for the implementation. One example is presented for each category. From the category component interface adaptation, the adapter layer type is presented, that allows the software engineer to change operation names at a component. From the component composition category, a layer type for delegating requests is presented. Finally, from the component monitoring category, the observer layer type is presented that can be used to adapt a component with observer functionality.

5.1 Changing Operation Names

The temperature sensor in the blood part of the dialysis system was not changed in the new version of the system. Consequently, the intention of the software engineers was to reuse the software component interfacing with the hardware sensor. However, the architecture was designed using an abstract sensor with an interface consisting of `getValue()` and `callibrate()`. Since the temperature sensor component was developed earlier, it’s interface consisted

1. We are currently working on a Java implementation of the compiler that generates Java output code.

of `readTemperature()` and `reset()` methods. The functionality of the methods was not any problem, only the naming of the interface.

The above is a typical reason for a component not to be reusable, i.e. incompatible naming of operations in the component. For this purpose, a design pattern *Adapter* has been defined by, among others [Gamma et al. 94], that converts the operation names used by clients into selectors that are understood by the component. In object-oriented languages, the adapter is implemented as an object that forwards (adapted) calls to the adaptee. The disadvantage of this approach is that the adapter generally is not reusable, it may suffer from the self problem and may result in considerable implementation overhead [Bosch 97].

One can recognise that a design pattern generally affects many aspects of the object behaviour, requiring the design pattern behaviour to be *superimposed* on the object. Within the layered object model, several design patterns can be represented as layer types. A layer of type Adapter is defined that provides the functionality associated with the design pattern. The Adapter layer can be used for class adaptation by defining a new adapter class consisting only of two layers. Adaptation at the object level can be achieved by encapsulating the object with an additional layer upon instantiation:

```
// object declaration
adaptedTempSensor : BloodTemperatureSensor with layers
    adapt : Adapter(accept getValue as readTemperature, accept callibrate as reset);
end;
```

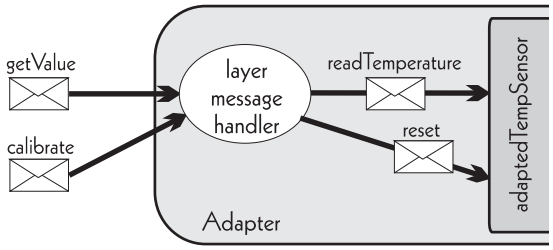

Figure 5. Changing operation names

In figure 5, the adaptation performed by the Adapter layer type is presented graphically. Messages with the `getValue` or `callibrate` selector are intercepted and changed to `readTemperature` and `reset`, respectively. Layer type Adapter can also be used in the inverted situation where a single client needs to access several server objects, but the client expects an interface different from the interface offered by the server objects. We refer to [Bosch 97] for a more extensive discussion on language support for design patterns.

5.2 Delegating Requests

The dialysis architecture uses the design rule that an actuator knows the result of its actuation. This requires a heater, for example, to be able to reply to a temperature reading request. In the dialysis system, a software component for controlling the dialysis concentrate pump was reused from an older version of the system. However, this component did not comply to the design rule. In order not to break the metaphor in the system, it was decided that the pump component would delegate `getValue()` requests to the concentrate sensor component which is placed right behind the pump in the system.

This example is a typical reuse inhibitor, i.e. the architecture expects a ‘larger’ interface from the component than the functionality provided by available individual components. Often a combination of components is able to provide the required interface, but since these components would lead to multiple entities to be invoked, this does not solve the problem. The most appropriate traditional solution is to implement a wrapper that encapsulates multiple components and forwards requests to the correct component, i.e. the *Facade* pattern. However, as we identified in [Bosch 97], the implementation of the facade pattern leads, among others, to considerable implementation overhead since the wrapper has to implement all methods that are supported by the wrapped components.

A solution to the problem is provided by *delegation*. Delegation provides object-level message forwarding whereas inheritance provides class-level message forwarding. However, traditional delegation-based object models, e.g. [Lie-

berman 86] and [Dony et al. 92], often require the delegating object to specify at definition time the objects that it delegated to, i.e. *explicit delegation*. This is infeasible for component adaptation since the intention is to reuse an existing component by adapting it to delegate requests to another component.

The solution available as part of **LayOM** is to extend an instance of the reused component with a `Delegate` layer. This layer delegates all messages that match the interface of the object that is delegated to. The object processes the request and returns the result to the `Delegate` layer that subsequently replies to the calling object. The message handling model in **LayOM** is such that *true* delegation is provided, i.e. `self` calls are sent to the original receiver of the message and not to the object processing the request. The syntax for the configuration of the `Delegate` layer is the same as that of the `Inherits` layer.

The code shown below illustrates the **LayOM** approach to the described problem. An instance, `dc_pump`, of class `DialysisConcentratePump` is extended with an additional `Delegate` layer. The layer forwards all `getValue` requests to an object `dc_sensor`, located in the context of `dc_pump`. As the '*' indicates, all other types of requests are not delegated. The example is graphically presented in figure 6.

```
dc_pump : DialysisConcentratePump with layers
    del : Delegate(dc_sensor, {getValue()}, * );
end;
```

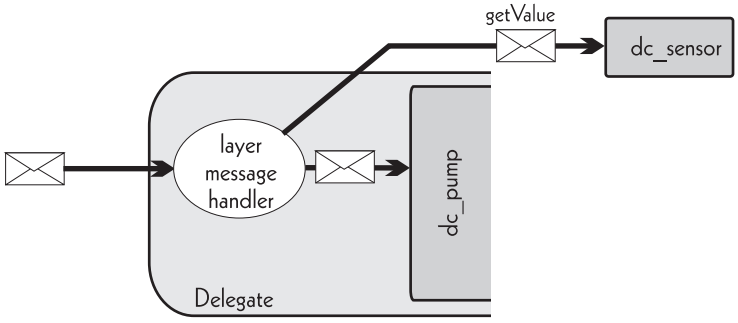

Figure 6. Delegating requests

5.3 Observer Notification

The unit for collecting the patient's accumulated weight loss is depending on the values read by the two flow sensors. Whenever one of these sensors reads a changed value, the unit has to be notified so it can adapt its behaviour. In the dialysis system, two instances of the same flow sensor component are used. The component is, obviously, not prepared for notifying any other components whenever it receives a new value from its corresponding hardware sensor. The observation behaviour has to be superimposed on the component.

The *Observer* pattern is widely used in object-oriented systems since is significantly decreases the dependency between an object and its dependent objects. In component-based systems, the pattern is very suitable for monitoring components for state changes. A problem is, though, that reusable components are generally not prepared for acting as an observable entity. The operations in the component do not provide the notification statements that normally are implemented in a component when defining it from scratch. Therefore, it may not be possible to use an otherwise reusable component in systems where the component should be observed by other components.

Similar to the previously discussed examples, the behaviour for component observation needs to superimposed on the component object. The solution in the context of the layered object model is to define a layer type `Observer`, that is used to extend a class to be used as a subject with behaviour for notifying dependent objects. Since layers intercept messages sent to and from the object and are able to inspect the abstract state of the object and notice state changes, an Observer layer is able to detect changes in the subject and notify the observants. The syntax of the layer is the following:

```
<id> : Observer( notify [ before | after ] on <mess-sel>+ [on aspect <aspect>], ... );
```

The layer intercepts messages and determines whether the selector in the message matches with one of the message selectors, `<mess-sel>`, specified in the layer configuration. If it does, the layer will notify the observants either before or after the message has been processed by the object. When the notification occurs depends on whether the `before` or

`after` keyword is used in the layer. Similar to the Smalltalk-80 implementation [Goldberg & Robson 89] of the observer pattern, the observant can be notified on a particular aspect, allowing the object to limit actual updates to only those aspects interesting for the particular observant. The administration of observer objects is also part of the functionality of the layer type. For this purpose, `attach` and `detach` methods are implemented in the layer and messages to the object with these message selectors will be intercepted by the layer and handled locally by the layer itself.

The `Observer` layer type can be used for extending a class with observer pattern functionality. The example class `FlowSensor` has, among others, a method update() that provides the updated value of the hardware sensor. Class `FlowSensor` is not observable, but the Observer layer added to the instantiation of the `FlowSensor` class allows other entities to observe the components's behaviour. The behaviour of the layer is presented graphically in figure 7.

aFlowSensor : FlowSensor **with layers**
    st : Observer(**notify after on** update **on aspect** "flow-value");
end;

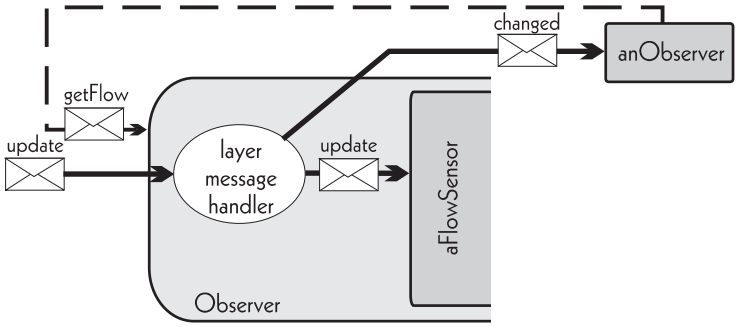

Figure 7. Observer notification

The `Observer` layer can both be used in class definitions and to extend individual instances with observing behaviour. Especially the latter is useful since it allows the software engineer to use a class without observer pattern functionality in a situation where it, among others, should play the role of a subject.

6 Related Work

The reuse of existing software is studied by the software reuse community and has lead to a substantial literature base. For instance, [IEEE Software 94] features a special issue on systematic software reuse and several reuse conferences, such as [Samadzadeh & Zand 95] and [Sitaraman 96], and overview papers, e.g. [Mili et al. 95]. Part of this research domain is the research on component-based software engineering. See, for example, the proceedings of the 1996 workshop on component-oriented programming in [Mühlhäuser 97]. However, the notion of adapting reusable components to match the requirements of the application at hand is not extensively studied in component-based software engineering. Some object models, e.g. CLOS [Kickzales et al. 91], provide *before* and *after* facilities that allow the software engineer to add pre- and post-behaviour to the execution of an operation in a component. [Purtilo & Atlee 91] introduce Nimble, a language that allows designers to declare how parameters in a procedure call are to be transformed in run-time to match the interface of the called procedure. Nimble, however, only deals with adapting the parameters of a procedure call, not with adapting the behaviour of a procedure. [Yellin & Strom 97] propose the software adaptors that are placed between two components that are functionally compatible, but have incompatible interfaces (or types). Software adaptors can be automatically generated from interface mappings, a high-level description means. Yellin & Strom's approach to software adaptors is different in several aspects. First, their adaptors act between two components, whereas layers adapt a component for all clients. Secondly, they use a high-level description language to specify adaptors, whereas our approach makes use of an extensible set of configurable component adaptors, i.e. the layer types. Thirdly, software adaptors are not transparent but clients communicate with the adaptor instead of with the intended component, whereas adaptation by layers is transparent and does not affect the communication between components. [Hölzle 93] discussed the problems of integrating independently developed components and concludes the traditional object-oriented languages provide little support. The only feasible black-box approach is wrapping but this leads to significant amounts of extra code and potentially serious performance problems. As a solution, *type adaptation* is proposed which allows the programmer to change the interface of a reused component after delivery. Compared to the work presented in this paper, type adaptation only deals with a restricted subset of the discussed adaptation types. Also [Samentinger 97] discusses component adaptation, and locates it between customiza-

tion, i.e. adaptation intended by the software designer, and modification, i.e. making major changes to the component. Adaptation is then categorised as minor modifications, which agrees to the approach taken in this paper.

The notion of superimposition has earlier primarily been used in the context of distributed systems, e.g. [Bouge & Francez 88] and [Katz 93]. There it is used to indicate the additional, superimposing control over some algorithm. To the best of our knowledge, our use and interpretation of superimposition for component object models is novel and solves important problems.

Since superimposition is a novel technique in object-oriented programming, no existing implementations of superimposition exist besides the layered object model. However, meta-object protocols [Kickzales et al. 91] can be viewed as types of superimposing behaviour for object-oriented systems. In general, reflective languages such as CLOS are suitable to implement superimposition. The composition-filters (CF) object model [Bergmans 94], being a partially reflective object model, provides some forms of superimposition. The CF model applies input- and output-filters that intercept incoming and outgoing messages, respectively. The CF model defines filter types like error, wait, dispatch, meta and real-time and each filter type provides one specific (orthogonal) type of behaviour. However, since the CF model provides, to the best of our knowledge, no means to extend individual instances with additional filters, the model can consequently not be used for component adaptation. Even if it would be possible, only a subset of the identified types of component adaptation would be supported.

7 Conclusion

Component-based software engineering is becoming increasingly important as a means to efficiently create applications from reusable components. Most traditional approaches assume that components are reused “as-is” in these applications, but in practice “as-is” reuse is very unlikely to occur and most components need to be adapted to match the requirements of the application. Five requirements for component adaptation were identified, i.e. a component adaptation technique should be transparent, black-box, composable, configurable and reusable. Conventional techniques for adapting components are copy-paste, inheritance and wrapping. The first two are white-box whereas the last is an example of a black-box adaptation technique. All traditional approaches fail to fulfil one or several of the identified requirements.

As a superior alternative, a new component adaptation technique, *superimposition*, was introduced. An object superimposition S of B over O is defined as the additional overriding behaviour B over the behaviour of a component object O. Different from, e.g. inheritance, a single unit of superimposed behaviour can change several aspects of the basic component’s behaviour. One of our conclusions is that, in addition to a set of reusable components, component-based software engineering requires a set of reusable component adaptation types. To exemplify this, several types of adaptation behaviour were presented and defined, categorised into component interface adaptation, component composition and component monitoring.

Superimposition is implemented as a language construct in the layered object model (**LayOM**) through the notion of layers. **LayOM** is an extended component object model that, next to instance variables and methods, contains parts such as states, categories and layers. The extended expressiveness of **LayOM** provides the software engineer with powerful component adaptation types through superimposition. To illustrate this, the implementation of three of the identified component adaptation types was presented.

References

[Bergmans 94]. L. Bergmans, ‘Composing Concurrent Objects,’ *PhD Dissertation*, Department of Computer Science, University of Twente, The Netherlands, 1994.

[Bosch 96a]. J. Bosch, ‘Relations as Object Model Components,’ *Journal of Programming Languages*, pp. 39-61, 1996.

[Bosch 96b]. J. Bosch, ‘Delegating Compiler Objects: Modularity and Reusability in Language Engineering,’ *Nordic Journal of Computing*, 4, pp. 66-92, 1997.

[Bosch 96c]. J. Bosch, ‘Object Acquaintance Selection and Binding,’ *submitted*, December 1996.

[Bosch 97]. J. Bosch, ‘Design Patterns as Language Constructs,’ *Journal of Object-Oriented Programming* (to appear), November 1997.

[Bouge & Francez 88]. L. Bougé, N. Francez, 'A Compositional Approach to Superimposition,' *Proceedings POPL'88,* pp. 240-249, 1988.

[Dony *et al.* 92]. C. Dony, J. Malenfant, P. Cointe, 'Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation,' *Proceedings OOPSLA'92*, pp. 201-217, 1992.

[Gamma *et al*. 94]. E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994.

[Goldberg & Robson 89]. A. Goldberg, D. Robson, *Smalltalk-80 - The Language,* Addison-Wesley, 1989.

[Helm *et al.* 90]. R. Helm, I. Holland, D. Ganghopadhyay, 'Contracts: Specifying Behavioral Compositions in Object-Oriented Systems,' *Proceedings OOPSLA '90,* pp. 169-180, 1990.

[Hölzle 93]. U. Hölzle, 'Integrating Independently-Developed Components in Object-Oriented Languages,' *Proceedings ECOOP'93*, pp. 36-56, 1993.

[IEEE Software 94]. IEEE Software, *Special Issue on Systematic Software Reuse*, September 1994.

[Katz 93]. S. Katz, 'A Superimposition Control Construct for Distributed Systems,' *ACM Transactions of Programming Languages and Systems,* Vol. 15, No. 2, April 1993.

[Kickzales *et al.* 91]. G. Kiczales, J. des Rivières, D.G. Bobrow, *The Art of the Metaobject Protocol,* The MIT Press, 1991.

[Lieberman 86]. H. Lieberman, 'Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems,' *Proceedings OOPSLA '86*, pp. 214-223, 1986.

[Mili *et al.* 95]. H. Mili, F. Mili, A. Mili, 'Reusing Software: Issues and Research Directions,' *IEEE Transactions on Software Engineering*, 21(6), pp. 528-562, 1995.

[Mühlhäuser 97]. M. Mühlhäuser (ed.), *Special Issues in Object-Oriented Programming*, dpunkt.verlag, 1997.

[Purtilo & Atlee 91]. J.M. Putilo, J.M Atlee, 'Module Reuse by Interface Adaptation,' *Software - Practice and Experience*, Vol. 21(6), pp. 539-556, June 1991.

[Reenskaug *et al.* 95]. T. Reenskaug, P. Wold, O.A. Lehne, *Working With Objects: The Ooram Software Engineering Method,* Prentice Hall, 1995.

[Samadzadeh & Zand 95]. M. Samadzadeh, M. Zand, *Proceedings of the ACM SIGSOFT Symposium on Software Reusability,* ACM Press, April 1995.

[Samentinger 97]. J. Samentinger, *Software Engineering with Reusable Components*, Springer Verlag, 1997.

[Sitaraman 96]. M. Sitaraman, *Fourth International Conference on Software Reuse*, IEEE Computer Society Press, April 1995.

[Yellin & Strom 97]. D.M. Yellin, R.E. Strom, 'Protocol Specifications and Component Adaptors,', *ACM Transactions on Programming Languages and Systems,* Vol. 19, No. 2, pp. 292-333, 1997.