Modelling Causal Connections Between Objects

Christer Lundberg
ARCS Research Group
Department of Technology
University of Kalmar
S-392 34 Kalmar, Sweden
Christer.Lundberg@te.hik.se
http://www.te.hik.se/~christer

Jan Bosch
ARCS Research Group
Department of Computer Science
University of Karlskrona/Ronneby
S-372 25 Ronneby, Sweden
Jan.Bosch@ide.hk-r.se
http://www.pt.hk-r.se/~bosch

Abstract
The conventional object-oriented paradigm provides client/server message passing as the primary means of interaction between objects. Although this fits well in those situations where one objects requests a service from another object, there exist other situations where one object, the *observer*, is depending on state-changes or actions occurring at an object, the *target*. In those situations, the OOP requires the target object to notify the observer objects, despite the fact that the target object does not benefit from the interaction. This inverted interaction scheme leads to problems such as increased coupling and decreased understandability, maintainability and reusability of the resulting classes. When analysing object interactions, one can identify four roles, the benefitor, the establisher, the sender and the receiver. The aforementioned problems result from the fact that in client/server interaction, the benefitor and the sender are not always the same object. To address these problems, we propose the notion of *causal connections*, an alternative interaction mechanism, complementing the traditional client/server interaction. Two implementations of causal connections are described; one in the context of C++ and another using the layered object model. It is shown that causal connections solve the identified problems.

Keywords

Object-Oriented Programming, Client/Server Interaction, Inter-Object Dependencies

1 Introduction

The *object-oriented paradigm* (OOP) uses message passing as a means for two objects to interact with each other. If object **A** wants to utilise a service at some object **B**, it sends a message to **B** requesting that service. Message sends are the only means for objects to affect other objects within the OOP. The object sending a message is normally referred to as the *client*, while the object receiving the message and performing the requested service is called the *server*. In applications, an object often acts both as a client and as a server, because it may need help from other objects when performing some service.

Objects that are pure servers, i.e. they don't request services from other objects, are normally easy to understand and maintain as reusable assets, since they do not rely on other objects for their behaviour. Therefore one is usually able to understand its behaviour completely by studying the class definition. If, however, an object also requests services from other objects, one has to study all these utilised services too, in order to fully understand the complete behaviour. Such objects depend on each other, and jointly contribute to providing the required service, even though the service may be accessible through the interface of a single object.

A service can be very complex and involve many co-operating objects. Several approaches have been proposed to represent a co-operating collection of objects as a single entity. Examples of such approaches are *design patterns* [Gamma et al.95], i.e. 'recipes' for solutions to common design problems, or *object-oriented frameworks* [Mattsson 96], i.e. structures representing adaptable applications.

The disadvantage of encapsulating collections of objects into black or grey 'boxes', is that it often is difficult to compose them with legacy software. This is due to the fact that the encapsulation prevents you from substituting some of the individual members in the collection with your own legacy objects. Frameworks offer some limited means for such replacements through their adaptation interface, but such replacements are not straightforward and introduce additional complexity. We refer to [Lundberg et al. 96] for a more detailed discussion.

Instead of modelling objects and their interactions as a single entity, there is a growing interest in modelling interactions separately from the object model. An interaction model can be viewed as analogous to the notion of a stage play, where the roles of the play are treated as requirement descriptions for the players (objects) that impersonate the roles. An object may take on more than one role, provided that it fulfils the specified requirements. Such interaction models have been referred to in current research as *role models* [Reenskaug et al. 96]. *Subjects* [Ossher et al. 96] are populated role models, using custom objects that all fit their roles. Others prefer to describe interactions in terms of *Contracts* between objects [Helm et al.-90].

Regardless of the interactions being treated separately or not, they are normally expressed in terms of the basic *client/server interaction mechanism.* Within the traditional OOP, an object can only affect other objects by making use of this mechanism. However, we have experienced that, in addition to the scenario where a client object requests a service at a server object, objects may depend on other objects, i.e. the state or behaviour of some *target* object is relevant to an *observer* object. We refer to the latter as a *causal dependency* between the target and the observer object. However, since the message send is the only means of communication, inter-object dependencies must be transformed into client/server interactions. This may cause several problems, such as increased coupling and reduced understandability, reusability and maintainability. This is due to the fact that target objects are responsible for managing the causal dependency, whereas this functionality logically belongs to the observer object.

To address these problems, in this paper, the notion of *causal connections* is proposed, a additional object interaction mechanism that provides direct support for causal dependencies. Although causal connections are implemented using message sends, they are provided to the software engineer as a conceptual modelling tool. In this paper, we propose two approaches to incorporating the causal connection in the programming language. The first approach is based on C++ and proposes a minor, uniform language extension and associated pre-processor to implement causal connections. The second approach is defined as part of the layered object model (LayOM), an extended object-oriented research language.

The remainder of this paper is organised as follows. In the next section, the notion of causal dependencies is defined in more detail, several examples are presented and the problems of traditional implementations of causal dependencies are identified. In section 3, the implementation of causal dependencies, the causal connection interaction mechanism, is described and it is demonstrated that causal dependencies are more easily implemented by causal connections than by client/server interactions. Section 4 describes C++ based implementation of causal connections, whereas the LayOM-based implementation is presented in section 5. Section 6 discusses related work and the paper is concluded in section 7.

2 Causal Dependencies

Objects in the real world affect each other by interacting in many different ways. In our applications we usually focus on cases where one object needs the help of other objects in performing its own tasks, e.g. a PID-controller requesting a valve to close itself in an automatic control program, or a simulation object requesting a database manager to store the collected information. It is rather straightforward to think of these cases in client/server terms. However, several types of interactions cannot be represented easily in client/server terms. In this paper, we refer to these types of interactions as *causal dependencies*.

In general, an interaction using messages can be seen as having four different roles:

- The **sender role** is responsible for sending the message.
- The **receiver role** receives the message and reacts upon it.
- The **establisher role** selects the sender and the receiver and establishes the connection between them, i.e. makes sure that the sender has a reference to the receiver when sending the message. This role is also responsible for all the work involved in selecting a sender and a receiver for the interaction.
- One can also identify a **benefitor role**. Usually the interaction satisfies the needs of some object, that may or may not participate in some other role. The object that benefits from the interaction is here referred to as the benefitor. The motives for the interaction can be found in the context of the benefitor.

If we look at traditional client/server interactions in terms of these roles, we find that all the four roles are taken on by only two objects.

- The client is the benefitor, the establisher, and the sender.
- The server is the receiver.

An interaction will be much easier to understand, if the establisher and the benefitor roles are taken on by the same object. In such cases, we can understand why that object does all the work to bring about the interaction. If they are different objects, the establisher's interaction effort gets confusing, because the reason behind it has to be found in another object. Such interactions thus reduce the understandability. It will, however, also reduce the maintainability, because one object does some work on behalf of another object, which introduces a tight coupling between them. Therefore, it is preferable to model the establisher and the benefitor together as a single object.

In a traditional object-oriented program, the code for establishing an interaction will normally be written in the sender object, because the actual message sending instruction has to supply all the information in the message and a reference to its receiver. This means that all this information has to be available to the sender object at that time, normally by having it stored in internal variables within that object. The encapsulation restriction prevents other objects from manipulating this information directly, which means that the management of this data also has to be done by the sender. Therefore, in a traditional object-oriented program, we will normally find the establisher and the sender roles assigned to the same object.

If the establisher role has to be assigned to the same object as the sender role, and the establisher role also should be assigned to the same object as the benefitor, there is only one kind of interaction that we are able to model accurately, i.e. the client/server interaction described above. There are, however, also numerous occurrences of interactions in the real world, where the receiver is both the establisher and the benefitor. In such cases, the roles will be assigned in the following way:

- Object A takes on the sender role.
- Object B takes on the roles benefitor, establisher and receiver.

This interaction pattern is often used when we want to make an object depend on events in another object, e.g. state changes or method executions. When translating this to an OO model, we have to assign the establisher role to object A, because of the semantics of message sending in the object-oriented paradigm. After the transformation, the roles will therefore be assigned as follows:

- Object A takes on the roles sender and establisher
- Object B takes on the roles receiver and benefitor

This will lead to the establisher and the benefitor roles being assigned to different objects, leading to decreased understandability and maintainability. In this paper, we introduce the concept of *causal dependency* for dependencies between real world objects, where the nature of the affected object and its actions give rise

to the interaction. In addition, we introduce an interaction mechanism, *causal connections*, for implementing such dependencies in a way that permits you to freely assign the establisher role to any object. By using the causal connection mechanism for causal dependencies, the establisher role can then be assigned to the same object as the benefitor role, regardless of whether or not that object is the sender.

> *A **causal dependency** from an object A, called the target, to an object B, called the observer, is a dependency which causes the observer to react on some event taking place in the target, without the target being affected by this process.*

We have selected the names *target* and *observer* for the roles in causal dependencies, because the interaction pattern is analogous to the real-world situation where someone (the observer) is observing someone else (the target of the observation).

2.1 Action-based causal dependencies

Action-to-action causal dependencies, i.e. causal dependencies where an action performed by the target triggers an action in the observer, are very common. From a naïve perspective, this seems like an instance of client/server interaction, but what makes it different is that the client is not the benefitor, and thus should neither be the establisher of the interaction. Often, the client is, conceptually, totally unaware of the interaction taking place at all.

Some years ago, the first author experienced a situation that can be used as an example of this type of dependency. For the first time in my life, he tried to 'invoke the *play* operation' of a fiddle that one of his friends had brought with him when he visited him. That action immediately caused the sleeping cat to execute its '*perform nervous breakdown*' operation and escape through the window. Everyone in the room were totally taken by surprise (probably also the fiddle), even if very few members of the audience did enjoy my performance.

At first sight, this may look like an ordinary client/server interaction where the fiddle scares my cat. There are, however, some important differences. When we map it into client/server interaction, we will be forced to make the fiddle responsible for deliberately scaring the cat. But why should the fiddle scare the cat? To the fiddle, it makes no difference whatsoever whether the cat continues to sleep or not. The nervous reaction of the cat is totally outside the fiddle's area of interest. We would therefore not expect the fiddle to provide for any reaction of the cat at all. Even worse, the friends in the room might also react in diverse ways to the sound, so the fiddle would have to keep track of them too and know what response to invoke in each person.

Such a fiddle would definitely surprise us, because the fiddle is normally just an instrument that emits sound when being played. It is not concerned at all about who is listening to it. The fiddle's additional responsibility to keep track of all the listeners and eventually notify them, indicates that the client/server interaction is unable to handle these kinds of situations appropriately. If we look at this in terms of the interaction roles, the fiddle has to be both the establisher and the sender. Whether or not the benefitor is the cat is difficult to decide, but it is certainly not the fiddle. According to our discussion above, the fiddle should therefore not take on the establisher role.

As another example of this kind of dependency, let us take a detective, that is hired by a very jealous husband to spy on his suspiciously happy wife. Assume that the detective is interested in knowing when someone invokes some of the wife's more delicate operations. In order to get activated, he must receive a message when one of these operations occurs. The wife is, of course, supposed to be totally unaware of the detective, so we will not expect her to help the detective in his observations. However, if we map this situation into client/server interactions, we will have to persuade the wife into helping the detective. In the real world, the detective is the establisher, but if we use client/server interaction, the wife must take over that role.

Action-to-state causal dependencies, i.e. causal dependencies where an action performed by the target brings about a state change in the observer, also occur frequently in the real world. When modelling this in a

system, action-to-state dependencies are transformed to action-to-action dependencies, since not doing so would force us to break encapsulation at the dependent object. The transformation is often made by bringing about the required state change through the invocation of a method rather than by manipulating the state directly.

A common example of action-to-state causal dependencies are counters that we use to register the number of times some event occurred. There is usually some property in the state of the counter that contains the number of occurrences encountered so far. When the action takes place in the target, that value must be incremented. As mentioned earlier, we will not modify the property directly, but instead define an appropriate operation for it.

2.2 State-based causal dependencies.

If we connect a cable between an amplifier output and a loudspeaker input, the result is that the electric current is forced to be the same on both connectors. When we disconnect the cable again, the dependency between the connectors also ceases to exist. In this example, the *output_current* property of one object is forced to be the same as the *input_current* property of another object. We refer to such dependencies as *state-to-state causal dependencies*.

Mapping state-to-state dependencies into client/server interaction is very difficult. Firstly, the state change in the client cannot by itself send a notification. Instead we must scrutinise all method code in the object and insert a message send after each instruction that modifies the state. Secondly, we do not want to modify the state in the server directly since that would break encapsulation, but instead we have to find a method that accomplishes the required state change. Due to these difficulties, state-to-state dependencies have been research topics for many projects. Some researchers are thinking of these dependencies in terms of *constraints* on the dependent properties, e.g. [Freeman 92, Sannella 93, Lopez et al. 94]. The *input current* property of the loudspeaker is then thought of as being constrained to have the same value as the *output current* property of the amplifier. We will discuss this more in section 6.

In many applications, a real world artefact is described from different perspectives, sometimes resulting in a separate object being defined for each perspective. Each perspective object reflects a specific role that the real world artefact plays in the application. Role objects need to be causally connected if they refer to the same real world artefact, since they generally share some properties. For example, one state element in one role object may correspond directly to part of the state in another role object. The problem of multiple models representing a single real world entity has been studied extensively within the relational database community, where different subsets of a relation are treated as separate relations, called *Views* [Date 86]. The same problem has also been studied in object-oriented research, e.g. in research concerning *Multi-View Objects* [Årzén 93] and *Subject-oriented programming* [Harrison et al. 93].

There are, however, also situations where the state change instead causes the invocation of an operation at another object. A very common example of this kind of *state-to-action causal dependency* is an alarm system. When the *measured_value* property in some temperature sensor exceeds a pre-set value, the alarm system is supposed to *raise* the alarm, e.g. by setting off a beeper or turning on a light indicator. In this case, the alarm system can be thought of as an observer, that is monitoring the state of a target sensor object. The observed object, i.e. the sensor, is normally not affected by the observation.

The distinction between an action or a state change as the triggering event in the target or as the response event in the observer may seem artificial. Normally, a state change in an object is achieved by invoking an operation in that object. Instead of using the state change as the trigger, we can just as well monitor the operation instead, activating the trigger when the operation is about to finish its work. We can normally also achieve a state change in some object by invoking some of its operations to bring about the required change. In fact, this is actually what we do in our causal connection implementation, because we do not want to violate the encapsulation of the observer object. When we look at real world phenomena, however, we normally think in terms of both state changes and actions, and consider both of them to be possible triggers and responses, so we have decided to keep the distinction here.

2.3 Problems using client/server interactions for causal dependencies

If we abide by the traditional object-oriented paradigm, we are forced to map causal dependencies to the only available client/server interaction mechanism. Doing so will, however, give an interaction where the establisher and benefitor roles are assigned to different objects, which will affect both understandability and maintainability in a harmful way.

If we, for example, map the detective/wife example into client/server interaction, and use the *Observer Design Pattern* [Gamma et al. 95] to implement it, the wife object has to be involved in the observation. She must be able to handle 'subscriptions' from all observers spying on her, so she is able to notify them when needed. In our experiences, we have found the traditional client/server interaction mechanism to be marred by the problems related to increased coupling and reduced understandability, maintainability and reusability. We believe these problems to have the following causes:

- **The target is assigned the establisher role**
  The observer, and not the target, is normally both establisher and benefitor in the real world. Using client/server interaction forces us to assign the establisher role to the target. This introduces a change in the semantics of the model that affects *understandability*, *maintainability* and *reusability*, in a harmful way, because;
  - The software engineer is forced to distribute code which logically belongs to the observer, and insert it into the target. This will increase the coupling between these objects.
  - The added code in the target will reduce understandability, because it does not belong to the target's area of interest.
  - The increased coupling between the target and the observer will reduce both the maintainability and the reusability of these objects, since aspects of the observer has been 'hardwired' into the code of the target.
- **The semantics of the target is obscured**
  The target will become a concept with two rather unrelated capabilities. For example, the fiddle has to become a 'cat scaring fiddle' in order to scare the cat, and the wife of the jealous husband must become a 'detective informing wife'. This has the following disadvantages;
  - The software engineer is forced to adapt reused components from, e.g. class libraries, in order to use them as targets, since reusable components are normally not prepared for interaction management.
  - The target will have rather unclear semantics, and will therefore be very difficult to understand, because parts of the situation in which it resides gets hardwired into the object.

3 Causal Connections

In this paper, the notion of causal connection is proposed as an alternative to the client/server interaction mechanism for implementing causal dependencies. We define the causal connection concept as follows:

*A **causal connection** from an object A , called the **target object**, to an object B, called the **observer object**, is an interaction mechanism where the occurrence of a **triggering event** in the target object causes an invocation of a **response operation** in the observer object. The object responsible for establishing a causal connection (and, consequently, for removing it), is called the **connection manager**.*

The **triggering event** may be one of the following:

- The target has received a request to perform an observed operation and is about to start the execution of it.
- The target has finished the execution of an observed operation.

A **response operation** is the operation taken by the observer in reaction to a triggering event that occurred in the target. It has the following characteristics:

- As an input, it takes a reference to the target object, the identity of the target method and a flag to indicate which trigger point that is active (i.e. whether the triggering event is before or after the execution of the operation).
- A response operation returns no value.
- It generally questions the target object for the information the observer depends upon by using ordinary client/server interactions.

As mentioned, four roles are involved in an interaction, i.e. the establisher, the benefitor, the sender and the receiver. It was demonstrated that the benefitor and the establisher roles should be assigned to the same object. As a result of this, the causal connection concept merges the benefitor and the establisher roles into the *connection manager* role. That role can be assigned to either the target or the observer, but it can also be assigned to a separate object.

The causal connection interaction mechanism has the following features:

- **A triggering event in the target sends a message to the observer**
  Causal connections deviate from the traditional client/server interaction pattern in the way the interaction is activated. In client/server interactions, the client is responsible for starting the interaction by sending a message to the server. In causal connection based interaction, the interaction is started implicitly. Whenever the triggering event occurs, a message is automatically sent to the observer.
- **The triggering event invokes the specified response operation in the o bserver**
  A triggering event causes a pre-selected *response operation* to be invoked in the observer object. The observer must therefore have the corresponding response operation available for invocation. All response methods must have a signature matching the definition. When establishing the connection, the connection manager selects which observer response method to be invoked.
- **(Almost) no special preparations are necessary for target objects**
  The target object should be unaware of the causal connection and, consequently, not have to make any preparations for it. The fiddle must be able to remain a usual fiddle without modification, but still be able to scare the cat. All wives that are being scrutinised by a detective, should be unaware of the observation. This means that any reusable component can function as the target object in a causal connection, without having to be adapted to do so.
- **Observer objects must have a response operation**
  As mentioned, objects acting as observers in a causal connection must provide the specified response methods. Returning to the detective and the wife, this agrees very well with our intuition that a detective usually has an observing capability, even when not under contract. This means that also a reusable detective class ought to have an observing capability.
- **Causal connections are dynamic**
  A causal connection may dynamically emerge and vanish during program execution. The connection between the wife and the detective emerges when the detective is hired, and ceases to exist when the jealous husband ends the contract. The connection manager can dynamically establish and delete causal connections during program execution.
- **Causal connections are independent of each other**

Existing connections to a target should not influence the establishment of a new causal connection. If many observers use the same target and perhaps even the same method in that target as a trigger, they must all be notified in some arbitrary order when the event occurs.

Causal connections provide a uniform mechanism for implementing all four aforementioned categories of causal dependencies. However, only action-to-action causal dependencies are implemented directly. The other causal dependency types are transformed into dependencies of this category. For example, if a state change acts as trigger, the event is generated at the end of each operation that modifies the state property. We assume conformance to encapsulation rules by the software engineers, so that no object state is manipulated by an external entity.

Causal connections provide two trigger opportunities for each operation in the target, namely when starting and when finishing execution. State change triggers are usually mapped into the latter, by activating the trigger points at the end of each operation that modifies the corresponding property. Triggering at the start of some operation is useful when the observer wants to insert behaviour before the operation performed by the target, e.g. starting a timer in order to time an operation. The trigger point at the end of the operation may in such cases be used to stop the timer.

The connection manager decides the response operation of the observer. Each trigger in the target is assigned a corresponding response operation in the observer. The observer may have different response operations for each connection, or it can use the same response operation for all its connections. Response operations are ordinary members of the observer object, and can therefore access its internal state. If the required response is a state change, the response operation must change the state as required.

3.1 Dealing with dependency chains and loops

In database systems, data dependencies between stored items are often causing considerable problems, since a database might easily end up in a state that violates the dependency rules. It is very easy to forget to update some dependent item in a database when an item is modified, thereby causing the stored information to become inconsistent. To prevent this, all dependency rules are stored in the database together with the data, and the database program is made responsible for abiding by these rules at all times. Experience has shown that it may be difficult to achieve acceptable performance in such a *Truth-Maintenance System* (TMS) [Rich et al. 91], especially if the number of dependencies is large, since even a small update may give rise to a chain reaction that can cause almost the entire database contents to be revised. When using causal connections, being somewhat similar to a TMS, one has to be aware of such performance issues.

A common problem, when transforming declarative statements into imperative algorithms, is the risk of getting circular dependencies. The declarative constraint statement ”F = 32 + 1.8*C” means that F depends on C, but it also means that C depends on F. If both dependencies are implemented by causal connections, this results in a loop. The compiler is normally not able to detect all loops, because causal connections can be established dynamically. There are, however, techniques to prevent them, e.g. by including a time tag in the messages that is remembered in each response method. If the same time tag arrives again in a following message, the message is simply discarded. In order not to further complicate the discussion in this paper, we do not go into further detail here. Instead we refer to, e.g. [Sanella 93], where *Multi-Way Constraint Solvers* are able to deal with this problem.

3.2 Implementation

In the subsequent sections, two implementations are presented. The first is based on C++ and uses a simple C++ precompiler, whereas the second implementation is based on the layered object model (LayOM). In the precompiler approach, the programmer just has to point out which methods in a class that are observable. This is done, in the header file, by assigning the keyword ‘*observable*’ to each such method. The precompiler will insert a ‘hook’ at the start and at the end of all the marked methods. In the second approach, the

layered object model, objects are encapsulated by a, possibly empty, set of layers. These layers act as wrappers and intercept all messages sent to and by the object. A target object can dynamically be extended with , a layer that manages the response messages to the observers, by intercepting incoming messages to the object.

4 Implementing Causal Connections using C++

In the C++ implementation, legacy library objects cannot be used exactly as they are, since we have to create a 'hook' in all methods that might be selected as triggers in an observation. To indicate what methods are potentially relevant for an observer, a new keyword *observable* is introduced. The observable keyword is applied to each method that we suspect may serve as a target in observations to come. We developed a precompiler that automatically converts the keyword into C++ code. Currently, the precompiler only deals with methods, but future implementations will also include property variables. Making a property variable observable will then be equivalent to making all operations that modify the property observable.

As noted before, the application of the new keyword does not make the class less reusable or context dependent, since selecting the methods potentially relevant to observers can be performed independent of specific applications. As an example, let us look at the wife again, and assume that our `Wife` library class looks like this:

```
class Wife {
public:
   …
   void move_yourself_to ( char* new_location )
   const char* get_location() const;
   const char* get_name() const;
private:
   …
};
```

There are three operations shown, that we have to consider as observable candidates. If we suspect that some object may need to be notified if the wife moves to another location, then we will mark the first operation as observable. This seems to be a reasonable assumption, so we do that. If we think that others might want to be notified if somebody asks the wife where she is, then we make also the second operation observable. The same goes for the third one. If we think that others might be interested in being notified if somebody asks the wife for her name, then we also make this third operation observable. In our opinion we consider only the first one to be interesting for others, so we choose only that one. The class definition therefore looks like this:

```
class Wife {
public:
   …
   observable void move_yourself_to ( char* new_location )
   const char* get_location() const;
   const char* get_name() const;
private:
   …
};
```

After inserting the keyword in the header file, that class is recompiled using the precompiler. The precompiler searches for the `observable` keyword in all class definitions. If one or more such keywords are

found in a class, it will insert some code in the header file and insert the hooks in the corresponding method definitions. After precompilation, the header file looks as shown below.

```
#include ”CC.h” // Contains all CC classes
extern const char* CClst_Wife[];
class Wife {
public:
   …
   void move_yourself_to ( char* new_location )
   const char* get_location() const;
   const char* get_name() const;
   int      CC_get_id   ( char* method_name ) const
      { return CC_vect.get_id ( CClst_Wife, method_name ); }
   CC* CC_activate ( CC* link, int id, int after )
      { return CC_vect.attach ( link, id, after ); }
private:
   CC_list CC_vect;      // ptr to CC_link chains
   …
};
```

The text added by the precompiler has been shown in italics. Most of this text is generic for each class, and a minor part of the text, printed in bold, is specific for the class that is extended. In the header file, only the name of the external constant gets a class specific name, i.e. `CClst_` appended with the name of the class. The `observable` keyword is removed in this process. The `CC_list` data member maintains a vector with pointers to `CC_links`, and the `get_id` operation for converting a method name string to an index in this vector, and an `attach` operation to insert and remove pointers in that vector. It also provides an `operator[]`, so that indexing can be done on the entire `CC_list` object and still refer to entries in the pointer vector.

The corresponding .C-file is extended with a definition of the constant vector `CClst_Wife`, and the ‘hooks’ are added at the start and end of the observable methods. The first hook will always be executed when the method is invoked, being the first statement. The latter hook is, however, not equally simple. There are many ways to return from a method in C++, by using the *return* statement in various locations in the code, that will prevent the execution to reach the last instruction in the method. Every return statement has to be preceded by a copy of the code for the last hook, so that it is impossible to return from the method without executing the hook code. Abnormal method termination, e.g. through exceptions or exits, are not incorporated in the model, but are target for future research.

The `CClst_Wife` vector contains an array of strings, where each string is the name of an observable method in the `Wife` class. These names are ordered so that the first encountered observable method in the class definition occurs first in the vector and so on. These names are used when some observer wants to connect itself to an observable method. A text string specifying the method name is sent in that call, and the vector is used for converting this name into an index in the `CC_vect` pointer list. The affected parts of the .C-file will look like the code shown below.

```
#include “wife.h”
const char* CClst_Wife[] = { “move_yourself_to”, (char*)0 };
…
void Wife::move_yourself_to ( char* location )
{
   if ( CC_vect[0] )  CC_vect[0]->CC_notify();
   < original method code >
   if ( CC_vect[1] )  CC_vect[1]->CC_notify();
}
…
```

The remainder of the code for management of causal connections resides in the CC base class and in the `CC_link` template, which is a subclass to CC. This code can be found in appendix A.

Rather than discussing the code in detail, it is discussed at a higher level. When a connection manager wants to establish a causal connection between some target object of class `TA` and an observer object of class `OB`, it creates a `CC_link<TA,OB>` object, i.e. it instantiates the `CC_link` template using both the target's and the observer's classes as template parameters. As parameters to the constructor, it sends a reference to target object, the name of the method it wants to use as trigger, and a flag indicating which of the two hooks in that method it wants the link connected to. The last two parameters are a reference to the observer an a pointer to the response method. The manager also sets up a pointer to the created link, in order to be able to delete it.

When the `CC_link` constructor is called, it will insert a pointer to itself in the `CC_vect` data member of the target object. If there is a `CC_link` already connected to the specified hook, it will consider it as the head of a doubly linked list, and insert itself at the top of that list, in front of the old link. Each hook in the target object may thus point to a doubly linked list of `CC_links`. When the hook is activated, it sends a message to the first link in the list, which passes it on to the observer. On return, the `CC_link` passes the message on to the next node in the list, which repeats the process until the entire list is traversed. A schematic description is given in figure 1. Each `CC_link` object contains a pointer to the observer response method which is not shown in the figure. The pointers from the observer objects to the links reveal that the observers have taken on the role as managers too in this example.

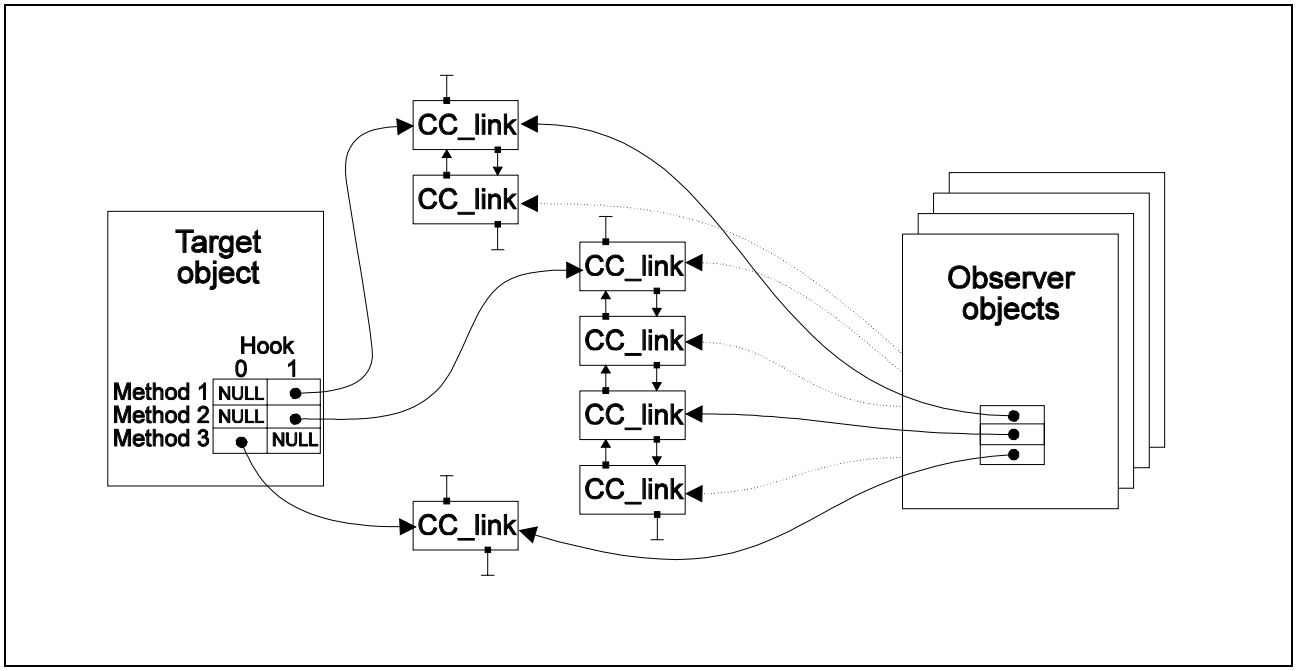

Figure 1: Causal connections between a set of observers and a target object

We have now seen how the wife is supposed to be prepared for being observed. The detective, in turn, has taken on the role as manager, and is therefore responsible for creating the link to the wife. This is done by supplying the needed parameters when the link is created. In our example, the detective creates the link as part of his `hire` method, that is invoked when some husband hires the detective to spy on his wife.

The code for the link creation might look like this :

```
int Detective::hire( Wife& wife, char* method_name )
{
```

```
    …
    _target = new CC_link<Wife,Detective>
        ( wife, method_name, 1 , *this  , &Detective::CC_response );
    …
}
```

The first parameter to the constructor is the target to spy on, which is given by the parameter to the enclosing method. The next parameter, also a method parameter, specifies the name of the target method to link to, further detailed by the third parameter, giving the number (0 or 1) of the hook within that method. The last two parameters are references to the observer and the response method to be invoked by the hook. When the detective wants to delete the connection, he needs only to delete the created `CC_link` object using the instruction ”`delete _target;`”. The `CC_link` destructor will handle the disconnection from the target and also do the unlinking from the list of `CC_links`.

Having prepared the detective for establishing causal connections, we have provided him with a capability to spy on any wife. A simple program using the detective together with a wife might look as follows:

```
#include “detectiv.h”
#include “wife.h”
int main()
{
   Wife       Laura;
   Detective  Mike;

   Mike.hire ( Laura, “move_yourself_to” );   // Let’s hire Mike to spy on Laura

   Laura.move_yourself_to(“PGA mall”);
   Laura.move_yourself_to(“Mr. Anderson”);

   Mike.print_report ( );
   Mike.fire ( );

   return 0;
}
```

In this case, the detective is hardwired to spy on objects from the `Wife` class. This can be avoided by making the detective class a template instead. In order to make the example as simple as possible, we have not taken that approach in our example. One can observe that

- The detective is doing all management for the connection, leaving the wife unaffected.
- The detective can at anytime establish and remove the link according to the actual assignments.
- The observing capability has been built into the detective. We can therefore think of the detective as an object that has an observing capability. This means that the detective object has some response method that enables it to act as an observer, and that it is able to establish and remove causal connections, i.e. to take on the manager role too.
- If causal connections were part of the basic OOP, the `Wife` class would have had its relevant methods marked as `observable`, since software engineers would be required to construct classes that way. In that case, the `Wife` library class could be used without any modification.

Let us also take an example of an alarm system monitoring a sensor. Again, the causal connection enables one to implement the system in an intuitive way. The alarm itself has the capability to be connected to a sensor and monitor its measured values. It assumes that all sensors have an observable method `read_new_value` that reads a new value and returns it to the caller, and a `get_value` method that returns the last measured value. A common base class `Sensor` could provide virtual versions of these methods, making it possible to connect an alarm to all kinds of sensors.

```
#include “alarm.h”
#include “sensor.h”
int main()
{
   Temp_Sensor  thermometer;
   Alarm        whistle;
   int          value;
   CC_link<Temp_Sensor,Alarm> link1
       ( thermometer,”read_new_value”, 1, whistle, &Alarm::check_value );

   while ( more_measurements_are_wanted() ) {
      value = thermometer.read_new_value();
      do_something_with_the ( value );
   }
   whistle.print_report ( );   // Has the whistle raised any alarms ?
   return 0;
}
```

In this example, the manager and observer are different entities, since an alarm system can be viewed as a rather lifeless object, that has to be connected to the device that is to be observed. The management is therefore taken on by the main function, that creates the link, i.e. `link1`. When the function is finished, the destructor is automatically called for `link1`, which will cause it to disconnect the alarm from the thermometer.

4.1 The pre-processor prototype

We have designed a first version of a pre-processor providing the functionality discussed above. The pre-processor employs a simple parser only recognising brackets, the *observable* keyword and few other tokens, rather than a full-fledged C++ parser. The pre-processor takes classes using the extended C++ syntax as input and generates correct C++ output by replacing the introduced new syntax with C++ code. The pre-processor first processes the class definition and subsequently the class implementation. While parsing the class definition, the parser searches for the *observable* keyword. If it finds it, it will remove the keyword from the output code, store the name of the method that should be observable and insert some C++ code in the class definition. For the class implementation, the pre-processor inserts some code at the beginning of the file and searches for methods that should be made observable. For each observable method, it inserts some code directly after the opening bracket and before each return statement in the method.

This first version of the pre-processor has a few limitations: nested classes cannot have observable methods, multiple methods with the same name but different argument types are not dealt with and target classes have to be processed before compiling observer classes.

5 Implementing Causal Connections using LayOM

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. In the previous section, the problems associated with representing causal dependencies between objects were discussed and a solution in the context of C++ was proposed. The C++ solution requires, on the one hand, minimal adaptations in terms of, among others, the programming language, but, on the other hand, additional efforts in the implementation of the causal connections. In addition, it requires the class to be prepared for observing behaviour since the `observable` keyword has to be applied for the relevant methods.

In this section, a solution to the problem of representing and implementing causal dependencies is described using the LayOM approach. The advantage of this approach is that the implementation efforts are minimised and that also unprepared classes can be used as targets. The disadvantage is, obviously, that LayOM is a research language without the industrial acceptance that C++ enjoys.

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 5.2, some example layer types are presented. Finally, in section 5.3, the specification of causal connections between classes and objects is described.

5.1 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 2, 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 97b] and acquaintance handling [Bosch 96b].

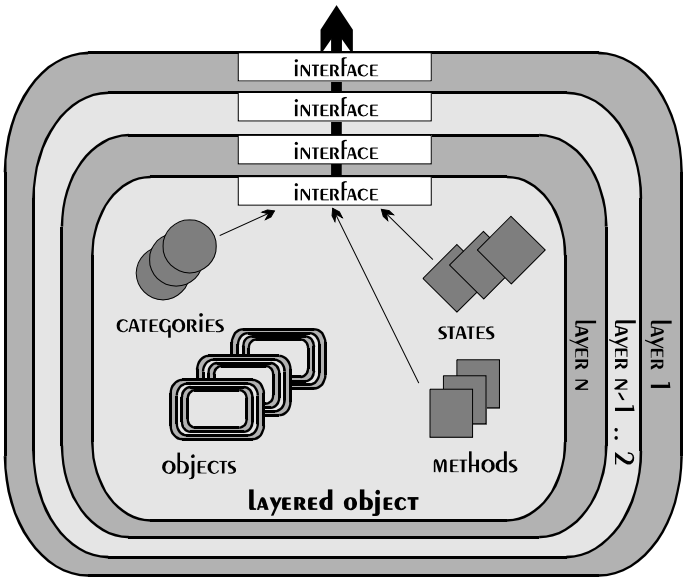

Figure 2: The layered object model

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.

A 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 categories to determine whether the sender of a message is a member of an acquaintance 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. Previously, layers have primarily been used to represent relations between objects, for the representation of design patterns and for acquainance 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 acquainance 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 LayOM compiler is based on delegating compiler objects [Bosch 97a], a concept that facilitates modularization and reuse of compiler specifications and extensibility of the resulting compiler.

5.2 Representing relations and acquaintance handling

The primary novel feature of the layered object model are the layers that encapsulate LayOM objects. Therefore, we discuss, in this section, two examples of layer types that have been defined earlier. The next section discusses the representation of inter-object relations, in particular partial inheritance, whereas acquaintance selection and binding is discussed in section 5.2.2.

5.2.1 Structural relations

Structural relation types, as described above, 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 Partial Inheritance 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 `TextEditWindow` contains a partial inheritance layer with the following configuration:

```
pin: PartialInherit(Window, *, (moveOrigin));
```

The semantics of the partial 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 `PartialInherit`. The layer type accepts three arguments. The first argument is the name of the class that is inherited from; `Window` 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: `moveOrigin`. The semantics of layer `pin` is that class `TextEditWindow` inherits the complete interface of class `Window`, except for `moveOrigin`.

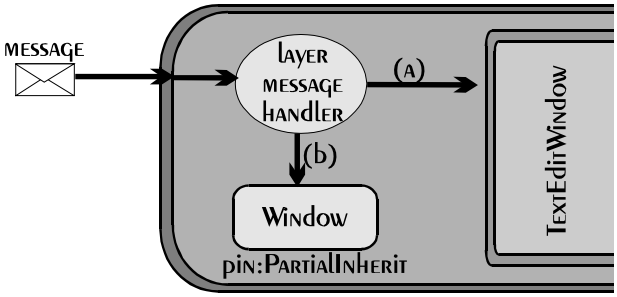

Figure 3. The `PartialInherit` layer.

In figure 3, the implementation of this semantics is illustrated. The partial inheritance layer is the second layer of class `TextEditWindow`. There is the most outer layer, shown around layer `pin` and an inner layer, shown around `TextEditWindow`. All inheritance layers create an instance of the inherited superclass, in this case an instance of class `Window`. 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 `Window`. 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 `Window`. If the selector is part of the set of interface elements, the message is redirected to the instance of class `Window`(situation (b) in figure 3). If the selector does not match with the interface of class `Window`, it is not redirected but forwarded to the next layer (situation (a) in figure 3).

5.2.2 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 we 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, introduced in the previous section, we have defined a new layer of type `Acquaintance`. 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 this in the layered object model, we discuss an example from mobile telephony. A physical mobile phone is represented by a mobile phone object in the distributed information system of a mobile telephony operator. A mobile phone object has a number of acquaintances, i.e. a base station object, an op-

erator object and a voice mail object. Since the owner of the physical phone moves around, the phone object also moves through the distributed information system. In some cases, the phone user travels outside the region covered by the operator, in which case another operator that covers the current region provides telephone services for the phone, under the condition that the two operators have such an agreement. Below, part of the LayOM specification for the mobile phone class is shown.

```
class MobilePhone
    layers
        base : Acquaintance(base-station is-bound per-call for one object from 1 contexts);
        oper : Acquaintance(operator is-bound from connect-operator until change-operator
                            for one object from global);
        mail : Acquaintance(voice-mail is-bound permanent for one object from global);
        ...
    categories
        base-station begin acq.subClassOf(BaseStation) and acq.owner = operator; end;
        operator begin self.operator = acq or acq.relatedOperator(self.operator); end;
        voice-mail begin acq.subClassOf(VoiceMail) and acq.phoneNumber = self.phoneNumber; end;
    ...
end;
```

The specification only contains the layer and category definitions for the aforementioned acquaintances of the mobile phone 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 `MobilePhone` class without the `Acquaintance` layers. That allows one to add the layer specifications to individual instances of the `MobilePhone` class, thus creating unique instances. For a more detailed description of acquaintance handling we refer to [Bosch 96b].

5.3 Representing causal connections

Causal connections, as described earlier, indicate a dependency of an observing object on some target object. State changes or the execution of actions may require related state-changes or actions at the observing object. For the observing object to know when it should change its state or execute its action, it needs to be notified by the target object. The alternative would be a very computationally expensive active polling solution. Traditionally, one would implement this using a variant of the *Observer* design pattern. However, as we identified in [Bosch 97b], this solution has several associated problems, such as the lack of traceability and the need for the target class to be prepared for observation.

A causal connection between objects should (1) traceable in the application, (2) separated from the target and/or observer classes and not mixed in the application code and (3) also instances of classes not prepared for acting as an observed object should be useable.

The solution in the context of the layered object model is to define a layer of type `Observer`, that is used to extend a class to be used as a subject with behaviour for notifying observing 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 et al. 89] of the observer pattern, the observant can be notified on a particular aspect, al-

lowing 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. The `attach` and `detach` methods are thus 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. Below the example class `ObservableWife` is shown. The class defines two layers; an `Observer` layer that notifies observing objects after a `move_yourself_to` methods has been executed and an `Inherit` layer that inherits all behaviour from class `Wife`.

```
class ObservableWife
   layers
      st : Observer(notify after on move_yourself_to);
      inh: Inherit(Wife);
end; // class ObservableWife
```

In addition to defining an `Observer` layer in a class definition, one can also extend individual instances with an Observer layer. This may even be more 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 target. Below, a instance of a regular `Wife` class without observer pattern behaviour is defined and extended with an `Observer` layer. This addition allows it to also be used as a target.

```
aWife : Wife with layers
   st    : Observer(notify after on move_yourself_to);
end;
```

The detective can be implemented as described earlier. The only action required from the detective object is that it registers itself at the target object, i.e. `laura.attach(self)`. The notification by the target is sent as an `anObserver.changed` message. If the observing object needs to be called at another method, it can use a layer of type `Adapter`. An example is shown below.

```
class Detective
   layers
      adapt : Adapter(accept changed as getLocation);
      ...
   ...
end; // class Detective
```

6 Related Work

The problems related to the client/server interaction mechanisms in the object-oriented paradigm (OOP) have been observed by several researchers. Some of them have focused on solutions that utilises the features that are already present in the traditional OOP, while others suggest extensions that they consider necessary. Most of the related work focuses on state-based causal dependencies, because such dependencies are very difficult to transform into ordinary client/server interactions.

A common case of state-based causal dependency are found in large applications where different teams develop different parts of the program. Very often, these program fragments represent different areas of interest, that all operate on a common real world domain. This means that the same real world entity may emerge in different guises in these fragments. This often means that one operation on an object in one frag-

ment of the program will affect the state of some object in another fragment, because they are models of the same real world entity. There are different approaches to deal with this dilemma. Either we look at different object models individually, and solve the dependency problem for each such object one by one, or we create all program fragments separately, and solve all the dependency problems jointly when the fragments are to be brought together.

6.1 Dealing with dependent objects individually

An approach belonging to the first category is to juxtapose all the views and create a giant object that reflects all them. This is usually accomplished by using multiple inheritance. Such objects have the advantage that all code and data about a real world entity is brought together in one place, but there are also disadvantages. Firstly, such giant objects with enormously rich interfaces will be very confusing to its clients. Think of the cockpit of an aircraft. If we were to put all these controls into a car, they would confuse the driver, even if you tell him that he can disregard 90% of them. Even if most of the controls are never used, they are confusing by just being there. Secondly, the propagation of state changes from one view to another has not been taken care of, but is still left to be solved, because different properties in the giant object may still depend on each other.

An example using 'giant objects' to support a set of views is given by [Årzén 93]. The basic idea is that each client uses a view of the object instead of the object itself. Each such view has a tailored interface that contains what the client wants to see and nothing else, but the real world entity is modelled as a single object supporting all these views. The views are ordinary objects that contain the necessary code to handle all propagation of state changes from themselves to other dependent views. This means that a client using a view can work with the view just like any ordinary object and totally ignore the dependency problems. When the different program fragments are brought together, the dependencies spanning the fragments are supposed to work already from the start, because all inter-object dependencies have already been taken care of at the object level.

A second approach to solve the dependency on the object level is to add other objects as helpers. In the *design pattern* catalogue [Gamma et al. 95] the authors describe an *Observer Pattern*, that is able to deal with different views on an object. The observer pattern solution is contained entirely within the traditional OOP, using only regular client/server interactions. It suggests that all shared properties are located in a host object, that is responsible for keeping track of all the dependent objects and for invoking an *update* method at all of them when some shared property has been modified. The dependent objects will thereby be able to keep their local states up-to-date.

A third approach, still operating on the individual object level, is to specify the dependent state of an object in terms of constraints. *Constraint imperative programming* [Lopez et al. 94] is an integration of declarative constraint-based programming and imperative object-oriented programming. Using constraints, the identity of some property can be set to the identity of another property, or a function thereof, which means that the dependent values thereafter abide by these rules at all times. In order to enforce this, there must be a dedicated constraint solver process that more or less runs in parallel with the application. The *SkyBlue Constraint Solver* [Sannella 93] is an example of such a solver.

Having the views as different objects has the same disadvantage that code and data belonging to one object is spread around the program. We have in an earlier paper [Bosch et al. 96c] described an approach where all this code instead can be kept together without producing the giant interface that was described earlier. Instead of satisfying everybody's needs with one interface, we use the layered object model (LayOM) to create an object that adapts it interface to the client interacting with the object, so that only those features are visible that the client wants to see. The views are then turned into layers that enclose each other like peels on an onion, and a message sent to such an object will automatically be delivered to the layer that agrees with the sender. All layers can interact with each other to achieve the propagation of state changes between them. LayOM can also handle other kinds of interaction besides pure state change propagation.

6.2 Dealing with dependencies when composing program fragments

In the second category, the dependencies between different views of objects, together with other dependencies are grouped together and transformed into the problem of making different dependent program fragments work together. In [Ossher et al. 96] each program fragment is considered to be a *subject*. According to their definition, a subject is ”an object-oriented program or program fragment that models its domain in its own, subjective way”. They use a *compositor program* to compose subjects into larger subjects. The detailed operation of the compositor program is controlled by a *composition expression* that has to be specified by the designer. Each subject consists of a *subject label* that describes the inner workings of the subject, together with the code (source or binary) belonging to the subject. All state dependencies between different views of an object are described in the subject labels of the corresponding subjects and will be used by the compositor program when the subjects are to be composed.

If we take all interactions between objects into account, instead of merely looking at interactions caused by dependencies between objects, some researchers try to separate them from the objects and create a separate interaction model. This can be thought of as describing the program as a stage play governed by a script that specifies a set of roles and what these roles are supposed to do at different times. The actors are then studied separately, and they participate in the play only if they take on some of the specified roles.

In [Reenskaug et al.96], the stage play script is called a *role model*. Each such role model specifies a set of roles that objects may take on. The authors have developed a method called *OOram* for working with such models. They have studied different ways of composing role models using *role model synthesis*. Because each role can be taken on by an arbitrary object, provided that it fulfils the requirements of the role, there is always the chance that one and the same object will take on more than one role. The OOram method is able to handle such cases, and is therefore also able to support different views of the same object. The main focus is, however, on the sequencing of operations to be performed in the composed model rather than on resolving causal dependencies between objects within that model.

A similar approach, focusing on the interaction between objects instead of the objects themselves, has been proposed in [Helm et al.90]. The authors call their approach *interaction-oriented design*, as opposed to the usual object-oriented design, because they treat interactions between objects as first class entities in the design space. They use *contracts* to define the behavioural composition of a set of communicating participants. A contract identifies (1) the participants and their obligations, (2) invariants that are to be maintained together with the actions that might be used to maintain them and (3) preconditions on participants to establish the contract and the methods that instantiate the contract.

Contracts can be *refined* by either specialising the type of a participant, extending its actions or deriving a new invariant which implies the old. Contracts can also include subcontracts, which makes it possible to build larger contracts from smaller ones. Maintaining causal dependencies between objects is probably possible if we use the invariant section of the contract to specify the causal dependencies and the actions to use when enforcing them.

7 Conclusions

This paper studies a class of inter-object dependencies, *causal dependencies*, that are difficult to transform into ordinary client/server interactions; the only interaction mechanism present in traditional OOP. The result of applying such a transformation on a causal dependency between two objects obscures the implementation considerably. This can be detected by looking at the code of the participants, where the code present in one object logically ought to be in the other object. This means that the object that has the motives for the interaction will not initiate it and that the object managing the interaction does not have any motives for doing so. Since an interaction can usually be understood by looking at the entire context but not by looking at the individual objects, the complexity of the objects involved in the interaction is increased considerably. ‘*How can one understand A by itself if its behaviour is influenced by B in some tricky way?*‘ [Meyer 88]. If the management code would be located in the object that has the motives, i.e. the benefitor, the client/server interaction mechanism will not be able to send any messages at all, because the benefitor is

in causal dependencies not the same object as the sender of the message. This causes problems related to increased coupling and reduced understandability, maintainability and reusability.

As a solution to the described problems, an interaction mechanism, *causal connections*, is proposed that extends the basic object-oriented paradigm. Causal connections are an alternative to client/server interactions and complements it. A causal connection can be considered as a relation involving three roles, the observer, the target and the manager. Different from the traditional client/server calling mechanism, causal connections provide the freedom to keep these roles apart and allows one to assign these roles to the appropriate objects. If the sender and the manager roles are assigned to the same object, the result is an interaction mechanism that is very similar to the traditional client/server interaction mechanism. However, if the manager and receiver roles are assigned to the same object, this gives an interaction that very closely maps to a situation where one object observes another object.

In this paper, two implementation approaches of causal connections are discussed. The first approach is based on C++ and defines a minor language extension, the `observable` keyword, that can be used as the public, protected and private keyword in C++. In combination with a pre-processor provides this approach a minimal extension, but major advantages in the support of causal dependencies. The second implementation is based on the layered object model (LayOM). A LayOM object is encapsulated by a, possibly empty, set of layers. One of the available layer types is `Observer`, that intercepts messages matching its specification and notifies observer objects that have registered themselves at the layer. Both approaches support causal connections and address the identified problems. The first approach requires more effort from the software engineer, but requires only a minimal extension to the industrially accepted C++ language. LayOM requires considerably less effort from the software engineer, but is an extended object-oriented language model that is compiled to C++. Since only limited effort has been spent on optimising the generated C++ code, the performance of the resulting application is less than what one would require from production-quality code. The main cause is the message handling by layers which is currently performed in a rather naïve manner, using message objects rather than the native C++ message passing semantics. Currently, a student project is ongoing to generate Java code, instead of C++ code. Part of the project is to optimise the implementation of message handling by layers, thereby improving performance.

References :

[Årzén 93] K. J. Årzén, 'Using multi-view objects for structuring plant databases,' *Intelligent Systems Engineering*, pp 183-200, Autumn 1993.

[Booch 94] Grady Booch, *Object-Oriented Analysis and Design with Applications, 2'nd edition,* The Benjamin/Cummings Publishing Company, Inc, 1994.

[Bosch 96a] J. Bosch, 'Relations as Object Model Components,' *Journal of Programming Languages,* Vol. 4, pp. 39-61, 1996.

[Bosch 96b] J. Bosch, 'Object Acquaintance Selection and Binding,' *submitted,* 1996.

[Bosch et al. 96c] J. Bosch, C. Lundberg, A. Hultgren, 'Multiple Object Interfaces in Object-Oriented Control Systems,' *Proceedings of ICECCS'96,* Montreal, Canada 1996.

[Bosch 97a] J. Bosch, 'Delegating Compiler Objects: Modularity and Reusability in Language Engineering,' *Nordic Journal of Computing* (to appear), 1997.

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

[Date 86] C.J. Date, *An Introduction to Database Systems,* Addison-Wesley, 1986.

[Freeman 92] B. Freeman-Benson, A. Borning, 'Integrating Constraints with an Object-Oriented Language,' *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 268 - 286, 1992.

[Gamma et al. 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

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

[Harrison et al.93] W. Harrison, H. Ossher, 'Subject-Oriented Programming (A Critique of Pure Objects),' *OOPSLA '93,* pp. 411-428, 1993.

[Helm et al.-90] R. Helm, I. Holland, D. Gangopadhyay, 'Contracts: Specifying Behavioural Compositions in Object-Oriented Systems,' *Proceedings ECOOP / OOPSLA '90*, October 21-25, 1990.

[Lopez et al.94] G. Lopez, B. Freeman-Benson, A. Borning, 'Constraints and Object Identity,' *Proceedings of ECOOP'94*, 1994.

[Lundberg et al.96] C. Lundberg, M. Mattson, 'Using Legacy Software Components with Object-Oriented Frameworks,' *Proceedings of the VITS-96 Conference*, University of Borås, Sweden, 1996.

[Mattsson 96] M. Mattsson, 'Object Oriented Frameworks, a survey of methodological issues,' *Licentiate Thesis,* Department of Computer Science, Lund University, Sweden, 1996.

[Meyer 88] B. Meyer, *Object-oriented Software Construction,* Prentice Hall, 1988.

[Ossher et al. 96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal, 'Specifying Subject-Oriented Composition,' *Theory and Practice of Object Systems*, Vol. 2(3), pp. 179-202, 1996.

[Reenskaug et al.96] T. Reenskaug, P. Wold, O. A. Lehne, *Working With Objects - The OOram Software Engineering Method*, Manning Publications, Greenwich, 1996.

[Rich et al. 91] E. Rich, K. Knight, *Artificial Intelligence (2nd edition)*, McGraw-Hill Inc., 1991.

[Sannella 93] M. Sannella, 'The SkyBlue Constraint Solver and Its Applications,' *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming,* MIT Press, 1994.

Appendix A

```
class CC {  // The CC base class provides the links
   public:
      CC() { prev=0; next=0; }
      virtual ~CC() { }
      virtual void CC_notify() = 0;
   protected:
      CC *prev, *next;
   };
```

```
   template <class Target, class Observer>
   class CC_link : public CC {
   public:
   //------------------------------------------------ constructor
      CC_link ( Target&   target,
                char*     method_name,
                int       at_end,
                Observer& observer,
                void (Observer::*response)( const Target*,int,int ) )
      {
         _target     = &target;
         _trigger_id = target.CC_get_id(method_name);
         _at_end     = at_end;
         _observer   = &observer;
         _response   = response;
         prev = 0;
         next = target.CC_activate(this,_trigger_id,at_end);
         if  (next) next->prev = this;
      }
   //------------------------------------------------ destructor
      ~CC_link ()
      {
         if (prev)
            prev->next = next;
         else
            _target->CC_activate(next,_trigger_id,_at_end);
         if (next)
            next->prev = prev;
      }
   //------------------------------------------------ callback
      void CC_notify ()
      {
         (_observer->*_response)(_target, _trigger_id, _at_end);
         if (next) next->CC_notify();  // Traverse down the linked list
      }
   //------------------------------------------------ private's
   private:
      Target*   _target; // Reference to target object
      int       _trigger_id;   // Index in target CC_vect for observable method
      int       _at_end; // Decides which hook to use in the method
      Observer* _observer;  // Reference to observer
      void (Observer::*_response)(const Target*, int, int); // Pntr to response method
   };
```