

Handling References With Aspect-Oriented Programming

Thesis

Written by: Ungár Péter

Consultant: Frigó József

Budapest, 2001.

Table of Contents

DIPLOMATERV FELADAT.....	4
NYILATKOZAT.....	6
OBJEKTUM-REFERENCIÁK KEZELÉSE ASPEKTUS-ORIENTÁLT ESZKÖZÖKKEL.....	7
HANDLING REFERENCES WITH ASPECT-ORIENTED PROGRAMMING....	8
1. CROSS-CUT PROBLEMS IN OBJECT-ORIENTED DESIGN.....	9
1.1. From Spaghetti Code to the Object.....	9
1.2. Overview of Object-Oriented Design.....	10
1.3. The Markings of Good Object-Oriented Design.....	11
1.4. Cross-Cut Problems.....	12
1.4.1. User Interfaces.....	13
1.4.2. Persistency.....	13
1.4.3. Distributed Computing.....	14
1.5. The Road to Aspect-Oriented Programming.....	14
1.6. References as Signals and Slots.....	15
2. CONNECT - A LANGUAGE FOR DEFINING SIGNALS AND SLOTS.....	17
2.1. Concepts.....	17
2.1.1. Signal.....	17
2.1.2. Connection.....	17
2.1.3. Signal Proxy.....	18
2.1.4. Adapter.....	19
2.1.5. Transition.....	19
2.2. Language specification.....	19
2.3. Support Classes.....	24
3. IMPLEMENTATION OF CONNECT.....	26
3.1. Implementation in C++.....	26
3.1.1. Creating slots.....	27
3.1.2. Adapters.....	29
3.1.3. Signals.....	30
3.1.4. Connect.....	30
3.1.5. Transitions.....	31
3.1.6. Weaving Steps.....	31
3.2. Implementation in Java.....	32
3.2.1. Creating slots.....	32
3.2.2. Adapters.....	33
3.2.3. Signals.....	33
3.2.4. Connect.....	33
3.2.5. Weaving steps.....	33
4. MIXING CONNECT WITH DESIGN PATTERNS.....	35
4.1. Structural Patterns.....	35

4.1.1. Adaptor.....	35
4.1.2. Decorator.....	38
4.1.3. Bridge.....	39
4.1.4. Facade.....	40
4.1.5. Proxy.....	41
4.2. Behavioral Patterns.....	42
4.2.1. Mediator.....	42
4.2.2. Observer.....	43
5. GENERSYS – AN EXAMPLE OF USE.....	45
5.1. Description of the Software	45
5.2. Original Architecture.....	47
5.3. Redesign with Signals.....	50
6. CONCLUSION.....	52
6.1. Comparison to Others.....	52
6.2. Future Improvements.....	53
7. ACKNOWLEDGMENTS.....	55
APPENDIX A. BIBLIOGRAPHY.....	56
APPENDIX B. FLOPPY CONTENTS.....	57

Diplomaterv feladat

Ungár Péter

szigorló mérnök-informatikus részére

Objektum-referenciák kezelése aspektus-orientált eszközökkel

Az objektum-orientált programozási paradigma gyengéje, hogy az alkalmazási területre vonatkozó részek összefonódnak technikai jellegű feladatokkal, mint az adatbázis-kezelés, vagy a grafikus felhasználói felület. Ezek a feladatok rendszerint nem rendelhetők egyetlen objektum hatáskörébe, hanem eloszlanak, és több objektum kooperációját igénylik. Az összefonódás rontja a szoftver karbantarthatóságát.

Ezek a hatások részben kiküszöbölhetőek egy olyan kommunikációs rendszer segítségével, amely lehetővé teszi objektumok kooperációját szoros csatolásuk nélkül. Ha a kommunikációt támogató részeket aspektus-orientált eszközökkel és egy speciális leíró nyelv segítségével illesztjük be a programba, a program karbantarthatósága javul.

A hallgató feladatai

1. Foglalja össze az aspektus-orientált programozás kialakulásához vezető lépéseket!
2. Definiáljon egy leíró nyelvet, amely alkalmas az objektumok közötti információterjedés leírására! A mechanizmus legyen általános, különböző objektum-orientált struktúrákhoz illeszkedő, és a lehetőség szerint programnyelvtől független!
3. Dolgozza ki a tranformációs mechanizmust, amellyel a kommunikációra vonatkozó programrészletek automatikusan beépíthetők egy programba! Implementálja a transzformációt egy programnyelvre!
4. Mutassa be, hogyan illeszkedik a kommunikációs rendszer tervezési mintákhoz! Hozzon általános példákat, és konkrét szoftver megoldást is!
5. Hasonlítsa össze a programtranszformációs mechanizmust más laza kommunikációt támogató eszközökkel, értékelje, és tegyen javaslatot továbbfejlesztési lehetőségekre!

A feladat beadási határideje

2001. december 14.

Tanszéki konzulens:

Dr. Frigó József

Számítástudományi és Információelméleti Tanszék

Záróvizsga tárgyak:

1. Adatbázisok
2. Matematikai logika
3. Tudásalapú architektúrák

Budapest, 2001. szeptember 30.

Dr. Frigó József
egyetemi tanár

Nyilatkozat

Alulírott, Ungár Péter, a Budapest Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Ungár Péter

Objektum-referenciák kezelése aspektus-orientált eszközökkel

Az objektum-orientált szoftver fejlesztés során foglalkozni kell egy sor technikai jellegű problémával (felhasználói felület, elosztottság, stb.). Ezen problémák kódja belekerül az alkalmazás program kódjába. A különböző területekhez tartozó kódrészek összefonódása rontja a program karbantarthatóságát. Ezen segít az aspektus-orientált programozás, amelynek célja a alkalmazás-specifikus programrészek és a technikai részek (az aspektusok) szétválasztása.

Ebben a dolgozatban egy olyan módszert és leíró nyelvet dolgoztam ki, amely egy aspektus-orientált keretrendszer alpjául szolgálhat. A nyelv az objektumok közötti kommunikációt támogatja anélkül, hogy a résztvevő objektumok egymásról tudnának. A nyelv megfelelő transzformációs mechanizmussal (ún. weaverrel) beépíthető tetszőleges objektum-orientált nyelvbe beépíthető. A dolgozatban részletesen a C++ nyelvbe való beépítést folyamatát írtam le, és érintettem a Java nyelvet.

A módszer fő előnye a különböző tervezési mintákat támogató konstrukciók.

Handling References With Aspect-Oriented Programming

In the course of object-oriented software development, a number of technical problems need to be dealt with (such as the user interface, distributed computing, etc). The source code for these problems mingles with the rest of the source code, making the objects less reusable and the code less maintainable. Aspect-oriented programming is set to help this by separating the application-specific code from the technical parts (the aspects).

In this thesis I will describe a method and a language which could be the basis of an aspect-oriented application framework. The language supports communication between objects without the objects having to know about each other. The language can be injected into existing object-oriented languages with a transformation mechanism, called the weaver. I will elaborate on the weaving process for C++, and mention on Java.

The greatest asset of my method are certain elements which support for design patterns.

1. Cross-Cut Problems in Object-Oriented Design

In this introductory chapter, I will describe the road which leads to Connect. Connect is the name of a language which I have designed to describe a method for communication between objects without tight coupling. The reason for this chapter is to show why Connect can be useful.

Software technology's quest for the Maintainable Software Code has started from *ad hoc* methods and lead us to high-level object-oriented languages. We have found that *any* software that has a regular life cycle, needs to be composed of **reusable parts**. Of course, if the software is a quick hack, a one-shot project, or something that will never *ever* need to be changed or reused, then such design is needless, and we might as well just write it any way we please, it's not going to matter.

Let's see what we can do when it *does* matter.

1.1. From Spaghetti Code to the Object

Software technology has evolved greatly since the first programs. The initial *ad hoc* methods produced **spaghetti code** [JAR95]. Also referred to as “kangaroo code” (referring to the many jumps in the code), spaghetti code can make programs very fast and efficient, allowing the programmer to use “programming stunts”, special little tricks that are designed specifically to make the program more efficient.

Spaghetti code is a fine so long as there is only one person working on the program, and the program is within a certain size and complexity. A particularly bright person could write a thousand lines of spaghetti code, and still have control and oversight over the entire program. (The thousand-line figure is an estimate, we're talking about software complexity, not size. I have written basic programs on C64 which have reached the memory limits of the computer.)

Spaghetti code does have some drawbacks, mostly related to a limit in complexity and maintainability. Modifying such code may require rewriting huge portions, “unfolding” some of it's tricks. Pushing a button somewhere in the code could make a bell go off elsewhere unexpectedly. The appearance of the first high level languages (e.g. Fortran and Algol) didn't help the phenomenon later known as the “software crisis”. If a programmer quit his job, his programs could no longer be maintained. Imagine reading and understanding a 100-page book with no chapters or paragraphs, just continuous text.

Eventually the concept of **structured programming** was invented [DAHL72]. Structured programming introduced a level of modularity, called **functions**. Functions isolated an operation from the rest of the code. This is good, because that particular operation could be written independent of the rest of the code, but also bad, because some of the neatest tricks of spaghetti programming could no longer be used. This trade-off, however, allowed several programmers to work on a single software, each developing a set of the functions independently. Libraries of functions were written, taking on several common tasks (such as I/O handling), so programmers no longer had to continuously reinvent the wheel. These libraries matured to quality code.

The greatest asset of structured programming was that it allowed greater tasks to be broken down into **steps**. The steps themselves could be broken down into smaller steps.

Huge operations could be refined into pieces that were easier to oversee. **Layers of operation** could be defined, with the highest layer containing high-level commands, such as “solve problem A”. The highest level would then call functions from the second layer, which would contain instructions like “do the first step of problem A”. The layers could then be inspected independently, without having to have insight into the entire program. This is the **top-down** approach, known from other fields of engineering. Structured programs could grow in size beyond the limits of spaghetti code, ten thousand lines of structured code are still maintainable. (Again, I am talking complexity, not size. With good design and components, programs can “be fruitful, and grow, and fill the memory and background storage”.)

Structured programming has its own limits too. The limits involved the connection between the collaborating layers: **complex data structures** which needed to be precisely defined. The specification of each layer had to be very strict, so as not to break the layer above or below. For the largest applications, the “cleanest” structured implementation was either inefficient, or broke the layered architecture at certain points. Transforming the data structures from layer to layer became a burden.

The most important concept of handling complexity was **abstraction** and **decomposition**. This eventually led to **modularity**. A module is a piece of code which is handled separately from the rest of the program. A module has its own source file, which can be compiled or replaced without changing the rest of the modules. Modules can be linked together into a program by a linker. The module has an **interface**, through which it connects to the rest of the world. The word 'interface' was borrowed from the hardware world, the definition of the properties of all incoming and outgoing signals. The module interface defines all the symbols in the module which are visible from the outside world. Modular structured languages include C and MODULA-2.

Decomposing a program into modules is not always easy; the developers need to find chunks which have high internal cohesion and low cohesion with other modules. This yields simple, easy and clear module interfaces. Sharing data structures between modules need to be minimized, though, to maintain the modules' independence.

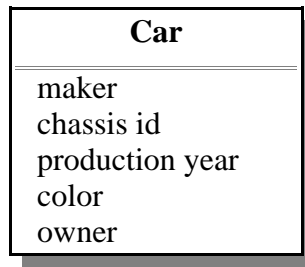
So far, the emphasis was on operations, and data structures were designed as the operations dictated. **Object-oriented programming** was a natural extension of modularity and the idea of abstract data structures. The concept is well known: let's place all the information about something in one place, along with all the functions that manipulate it, and we get “intelligent” data structures. Several definitions for the “object” exist, definitions that describe the concept of the object from different angles. Our definition shall be: The object is an entity that has a **state**, a **behavior**, and an **interface**.

1.2. Overview of Object-Oriented Design

Object-oriented software is said to **model** the real world [KON97]. Through a design process OOP methods analyze the problem domain, identify objects at a certain granularity, and create an object hierarchy. OOP is our means, not our goal. Our goal is creating reusable code. In this chapter I will summarize how OOP can help us.

There are three aspects of object-oriented design: the object model, the dynamic model and the functional model.

The **object model** emphasizes **data**. Each object class is described by its **class name**, **attributes** and **associations**. (Inheritance, subclasses and implementation pseudo-code can also be considered a part of the object model.) An object has three kinds of attributes: **identifying**, **descriptive** and **reference**. Identifying attributes are used to distinguish the object from others, such as a serial number for cars, and as such, seldom change their values. Descriptive attributes are internal properties of the object. Events can cause descriptive attributes change. Last, references are attributes which refer to another object, usually some kind of identifier or a pointer.



In the example above, the class *Car* has five attributes. *maker* and *chassis id* would be identifying attributes, *production year* and *color* are descriptive, and *owner* is a reference to a person who owns the car.

The object model is typically represented with a **Class Diagram**. Class diagrams contain Classes (such as the example above), associations between classes (the references between objects of the given classes), subclasses and maybe even implementation pseudo-code.

Second of the OOP models, the **dynamic model** describes the object hierarchy's behavior in time. The dynamic model shows how an object can change it's internal state, and also how objects communicate with each other. Collaborating objects typically send each other **events** (messages) via their references, which means that messages can travel wherever class associations are present. (Actually, Chapter 2 will be all about proving this false.) This is usually drawn as a **Communication Diagram**.

Last, the **functional model** describes data flow for the entire system. This model describes the system in terms of **inputs**, **outputs**, **processes** and **data stores**.

Together, the three models describe and object-oriented application from different aspects.

1.3. The Markings of Good Object-Oriented Design

The goals of Object-Oriented design are:

- Minimizing the cost of development (both present and future)
- Finding a good compromise between simplicity and flexibility

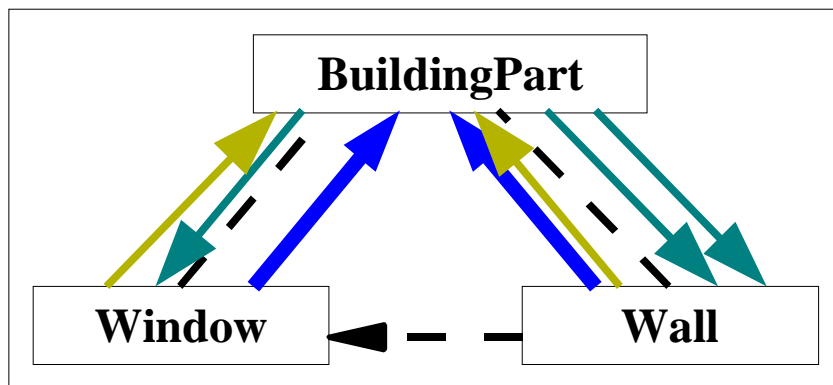
Here we have reached to a crucial point: what makes a design flexible? Experience has shown that designing *reusable* objects is very important [GAM95]. Designers need to find a solution that is specific to the problem, yet general enough to address future problems and requirements. However, to truly achieve this, designers would need to

“divine” the future of the software, wouldn't they? Or is there something else that one can do to make future changes easier?

First, let's consider we have to change, **standalone** class. By “standalone” we mean that the class doesn't depend on other classes, and there are no other classes which depend on it. The class still has a clearly defined purpose, and a class interface, we just don't need to think about that.

When we make changes to this standalone class, we can do anything we please, we don't need to consider side-effects that may arise from the change in the class' behavior. We could even change the class interface without the penalty of having to adapt other classes to the changes we make. Indeed, this is the ideal world, OOP heaven!

Next, let's take a look at a problematic design fragment [Ciu01].



Wall inherits BuildingPart. BuildingPart depends on Wall. Changes to either **Wall** or **BuildingPart** may lead to changes in the other. Same holds for **Window**. Further classes depend on these three, too, making this structure **resistant to change and reuse**. These three classes may be transparent enough, but extrapolated to 100 interconnected classes, the number of dependencies could grow exponentially.

The **communication model** can become quite complicated too, as soon as 10 or more objects' collaboration is required for handling a request. The number of messages can exceed the number which a developer can oversee.

Clearly, the need to eliminate unnecessary dependencies is great. In order for a design to be flexible and reusable, its' dependencies need to be minimized.

1.4. Cross-Cut Problems

One of the greatest dangers to clean design are **cross-cut problems**. Cross-cut problems are best described as tasks that are hard to assign to a single object, and are instead spread throughout the entire object hierarchy. This is very bad for reuse because

- Cross-cut problems require increased collaboration between object classes.
- This in turn creates a *complicated communication model*, and *convoluted object interfaces* which make the object hierarchy resistant to changes. (This is sometimes referred to as a “spaghetti interface” [JAR95])
- Methods for dealing with cross-cut problems are not part of the standard OO methodology [Kic97].

We refer to the task which the application has to solve as the **domain problem**. Some cross-cut problems are not domain specific, and are found in most modern applications. Let's consider some of these cross-cut problems.

1.4.1. User Interfaces

Most application programs have a user interface through which the user can get information and initiate operations. Graphical user interfaces typically run in an event-driven environment, such as the X Window System, Mac OS or Windows. The user interface can be identified as a cross-cut problem because it's shared among many objects, and can easily creep into the domain objects. However, the UI is not part of the domain problem, and is therefore a cross-cut problem.

Graphical objects in the user interface need to communicate with both the environment and the objects in the domain problem. The GUI may be completely separated from the domain, it could be an entirely different program on a different computer. Still, in most cases the GUI needs to be informed of changes in the domain to refresh itself, and this is where the extra references and interface element creep into domain object classes.

The number of techniques that deal with the separation of GUI from domain is great. The best solution should be efficient, easy to code and maintain. Some of the well-known techniques include the *Document-View Architecture*, the *Observer pattern*, or *Component-based Systems* (like Enterprise Java Beans [Mon00]).

To show just how hard dealing with GUI is, about half the patterns in the book "Design Patterns" [GAM95] either directly deal with the GUI problem, or use the GUI problem as an example.

However, attaching a GUI to an *application facade* is relatively easy. If we had an easy way to create and maintain an application facade *without* directly modifying the domain model, the user interface would cease to be a cross-cut problem. In Chapters 4 and 5 I will show how Connect can help.

1.4.2. Persistency

Objects in the domain problem need to be saved and loaded. This operation can be as simply as invoking a menu item from the File menu, or it can be a more intricate thing, such as unloading/reloading unused and unmodified objects to reduce memory usage. Objects can be serialized (their attributes stored linearly), saved to a database, a binary file format or an XML document. For the many forms of persistency, they are never part of the domain problem, often find their ways into domain objects, and are therefore cross-cut problems.

On the other hand, the code which implements persistency is often very straightforward, and writing it is rather simple. If we had a mechanism that transparently injects this code to the domain, persistency would no longer be a cross-cut problem. Java's implementation of serialization is a good example of an implementation which is transparent enough, so long as you serialize and de-serialize all your objects at the same time.

1.4.3. Distributed Computing

Separation of the application into client and server parts is a common practice. RMI, CORBA, and DCOM are only a few of the distributed technologies upon which such applications can be built. However, the *fact*, that the application has a distributed architecture produces a huge amount of code in all objects which use objects on the “other side”. This code mainly deals with establishing and maintaining a connection, authentication, synchronization, etc. Since all this code mixes with the domain code, making it another cross-cut problem.

The code introduced by distributed computing is, for the most part, trivial in the sense that there are a few patterns that need to be repeated over and over, patterns which depend little on the context (they depend more on the technology). If we had a way to insert this code automatically, distributed computing would no longer be a cross-cut problem, and at the same time our code would become *independent* of the technology.

I will mention in Chapter 6 how Connect can be extended to handle network transparency.

1.5. The Road to Aspect-Oriented Programming

The premises:

1. Object-oriented design is best when the model is generic, yet concrete to the task. Objects need to be reusable.
2. Object reusability is diminished by dependencies. The fewer the object's dependencies, the more reusable and maintainable it is.
3. Cross-cut problems introduce dependencies at all levels.
4. The cross-cut problems outlined above introduce code and dependencies that could be generated automatically.

The concept of **aspect-oriented programming** [Kic97] is **separating** the domain model from the cross-cut problems. The code which deals with them is **inserted automatically** by a **weaver**, based on information stored elsewhere. The information could then be stored in a language (which is recognized by the weaver), specialized to the problem at hand, and therefore much more *straightforward and concise*.

Together, the domain code, the **aspect-description language(s)**, and the weaver(s) produce an application which has all the features of the traditional OOP application, with the benefits of *less and more transparent source code*. Less and more transparent source lead to *fewer dependencies* and ultimately reusable objects and a well-maintainable application.

The strength of OOP lies in the OOP application frameworks which take on the burden of low-level programming. AOP is a new technology, and there are no AOP frameworks yet. In the future, AOP frameworks could have all the good things of OOP frameworks, with additional benefits.

As soon as AOP frameworks become as widespread and accepted as OOP frameworks are now, they will replace the previous paradigm.

In the next chapter I will outline a technique which could be the basis of an AOP framework.

1.6. References as Signals and Slots

References between objects define a relation between them. Objects can easily collaborate with other objects, objects that they have references to. Collaboration with other objects is possible using *lookup methods* or *mediator objects*, but references are the most common. Indeed, references are often important attributes of objects. For example, a scrolling area of a window has references to its scroll bars. If these references are null, the area is not scrollable. Or a **Dog** object may have a reference to a **Person**, called **owner**. The owner object is the responsible for feeding the dog. If the owner field is null, the Dog will be a stray.

References are sometimes stored as pointers, sometimes as something more abstract, such as an Object ID. The representation itself isn't as important as the fact that references tightly couple two objects. **Tight coupling** is fine so long as both objects are in the same domain (two objects dealing with the user interface, or two objects in the domain problem, etc). However, tightly coupling objects in different domains, is both bad design (it introduces interfaces and dependencies between domains), and very hard to avoid.

In the following chapter, I will introduce an Aspect-oriented approach to dealing with dependencies between domains.

As an example, imagine that there is a **Thermometer** class, which does some hardware-related operations to read the temperature of the system's CPU's. We also have a **CpuGuard** class, which reads the temperature from the Thermometer, and if it reaches a certain level, it sets an alarm, and halves the CPU's clock frequency. On the user interface side, we have an **IconLabel** class which can display an icon, and a **TextLabel** class which can display text. We would like an IconLabel to appear then the CpuGuard sets the alarm (showing a warning sign, or something), and the TextLabel to show the actual temperature in °C. Some or all of these classes could be part of one or more object-oriented class libraries (for example, we seldom write our own basic GUI widget classes).

One approach would be to subclass Thermometer to inform the CpuGuard of the changes in temperature and to inform one TextLabel to display the current temperature. The CpuGuard would be subclassed to inform the IconLabel. At this point we have introduced four dependencies in two objects: CpuGuard depends on IconLabel and Thermometer, Thermometer depends on TextLabel and CpuGuard.

Another approach would be to create a mediator class, **ThermoMediator**, which manage the distribution of information from the Thermometer to the CpuGuard and to the labels. ThermoMediator would depend on all of the objects, but it is simple enough to be maintained, and the only overhead is the introduction of an extra object to the class hierarchy. On the other hand, ThermoMediator's code is very simple. It needs to acquire some information from the Thermometer, then pass it on to a TextLabel and the CpuGuard, and finally call a method in IconLabel depending on the state of CpuGuard. All this code is simple enough to be *automatically generated*.

The injected code should notify certain objects when the state of another object changes.

In the next chapter, I will introduce a language, Connect, for describing signals and slots in a platform-independent way. In order to support complex designs (e.g. design patterns), a few additional concepts (such the signal proxy) will be defined.

In Chapter 3, I will demonstrate how this language can be used to insert code into two specific languages: C++ and Java.

Chapter 4 will describe how the features of Connect can be used in implementing abstract design patterns.

Chapter 5 will demonstrate how applying Connect can help make the design of a real-world application more flexible.

2. Connect - A Language for Defining Signals and Slots

Connect is a language suitable for describing abstract signals. Connect is independent of the underlying OOP language, it is built on the concepts of object-oriented programming: classes, objects, methods, etc. Some of the features of Connect may not be available under all languages; for example, if the language doesn't have method overloading, then overloaded signals cannot be used for that particular language.

Connect can be applied to an OOP language with a **weaver** specific to the given language. I will focus on C++ and Java™, but it would be interesting to see Connect implemented for others, such as Perl. C++ has many proprietary extensions, some of which implement similar functionalities as Connect. These extensions, however, are platform- and vendor-dependent, so it isn't always a good idea to use them.

2.1. Concepts

2.1.1. Signal

A **signal** is a sign that the object's internal state has changed in some way which may be important to the outside world. Zero, one or many objects can listen to a single signal. A signal is always the property of an object. A signal has a signature similar to that of a method:

- The signal has a **name**. The rules for signal names are the same as the rules for method names (the name must be unique, only alphanumeric characters can be used, must start with a letter or underscore).
- The signal can have **parameters**. Signals can be overloaded, the same signal name can take different parameters, if the underlying OOP language supports method overloading. The signal's parameters are transmitted to every listener.
- The signal always has a **void return type**.

The Thermometer example class (from Chapter 1.6) could have the following signals:

```
/* Signal is emitted whenever the temperature changes */
void temperatureChanged()
/* Same signal, but the actual temperature is passed, too */
void temperatureChanged( double temperatureInCelsius )
/* Similar signal, but the actual temperature is passed in
Fahrenheits instead of Celsius */
void temperatureChangedF( double temperatureInFahrenheit )
```

2.1.2. Connection

A signal can be connected to a number of **slots**. A slot is a *callable*, as defined by the underlying OOP language. Usually, callables are methods of objects, class methods (static methods), or functions (in C++).

The rules for connections are:

- The slots may have a **return type**, the returned value is **ignored**.
- A signal can be connected to many slots, and many signals can connect to one slot.
- A signal can be connected to and disconnected from slots at runtime.
- A signal can only be connected to a slot which has the *same signature*. For example, a `signal(double, bool)` can be connected to a `double slot(double, bool)`, but not a `void slot(int, bool)`, because there's no automatic type conversion. Subclasses of objects can be used.

For example, the `CpuGuard` class from chapter 1.6 could have the following method:

```
bool newTemperature( double temperatureInCelsius )
```

The `Thermometer`'s `temperatureChanged(double)` signal can then be connected to this slot. The objects no longer need to 'know' about each other, all the required communication is taken over by `Connect`'s signal/slot mechanism.

Our slots are in fact similar to callbacks, except that a slot can call a specific object's method, and can be adapted (see below).

2.1.3. Signal Proxy

A **signal proxy** is an object which is made to copy all of the signals of another object. Any signal emitted by the proxied object is re-emitted by the signal proxy. The rules for signal proxies are:

- Any object can be made the signal proxy of another.
- Signal proxies are defined and deleted at runtime.
- Signal proxies can be **stacked**: if you make object B the signal proxy of object A, and object C the signal proxy of object B, then C will emit object A's signals as well.
- An object can have multiple signal proxies
- An object can be signal proxy for multiple objects. If these objects have signals with the same signature, the signals are *merged*.
- The signal proxy relationships **can be cyclic**. This means that two (or more) objects are signal proxies of each other, either directly or indirectly (through stacking).
- Signals emitted appear to come from the proxy, not the original object.

Suppose that the designer of the `Thermometer` class chooses to use the Bridge design pattern [GAM95] to decouple the `Thermometer` abstraction from its' implementation. The `Thermometer` would no longer have any internal state, only an implementor, called `ThermalImplementor`. The `ThermalImplementor` would surely have a `double` `temperature` property, and a signal, `temperatureChanged(double)`. This, however, does us no good, because `CpuGuard` and others cannot connect to this 'hidden' class. However, if we make every `Thermometer` object the signal proxy of their implementors, the desired effect is achieved: `Thermometer` will appear to have a `temperatureChanged()` signal.

As I will demonstrate in Chapter 4, signal proxies are extremely useful, and make `Connect` more powerful than other tools which implement signal/slot mechanisms.

2.1.4. Adapter

Sometimes the problem domain's objects and other domains' objects use *different types*. Suppose there's a CAD program, which stores coordinates in double, and a widget set which uses integer coordinates. Signals from the domain objects cannot be directly connected to the widget set. Suppose the domain has a class, `Line`, which has four double attributes, `x1`, `y1`, `x2`, `y2`. Whenever `x1` or `y1` changes, `Line` emits a `startPointChanged(double, double)` signal. The widget set has a corresponding object, `CanvasLine`, which has a `setStartPoint(const Point&)` method. `Point` can be constructed from two integers. It would be good to connect these, but we can't, because the types don't match.

Enter **adapters**. Adapters have a static method which takes the signal's parameters, modifies them, and calls the slot with the adapted parameters. In effect, the slot will appear to have changed its signature. The signal will appear to have come from the originator, not the adapter.

Continuing the example above, we could have an adapter, `CADPointToGUI`, which could convert a `(double, double)` to a `Point`. We can now connect our `Line` objects to `CanvasLines` directly.

Adapters can accomplish more than this: for example, they could be used to provide *translations* (a string to string adapter which takes messages from one component and passes them on to another in a different language). This is useful if we are using a backend class which comes in English only, and want to use it in an international application.

2.1.5. Transition

Transitions implement Petri-net like transitions.

- Every kind of transition has a distinct class name.
- Transitions have two or more slots.
- When all of the slots have been activated, the transition fires a single signal, called `activated()`. The transition's signal's signature is the slots' signature concatenated. For example, if the transition has three slots: `slot1(double, bool)`, `slot2()` and `slot3(Line&, int)`, the signal's signature will be `activated(double, bool, Line&, int)`.

Transitions are useful for dealing with asynchronous, independent, cooperating subsystems. They aren't strongly tied to the rest of Connect's concepts, and are in fact more of a demonstration of how Connect can be extended to do more than relaying messages between objects.

2.2. Language specification

Connect is an **XML-based markup language**. The document type string is "CONNECT".

The following tags are defined:

CONNECT – The main tag

<i>Syntax</i>	<CONNECT> ... </CONNECT>
<i>Attribute specifications</i>	• VERSION= <i>string</i>
<i>Contents</i>	Exactly one PLATFORM tag, followed by zero or more CLASSES, ADAPTERS, and TRANSITIONS tags.
<i>Contained in</i>	Connect is a top-level element

The **CONNECT** element is the main element of a connect file. It can contain multiple **CLASSES**, **ADAPTERS** or **TRANSITIONS** tags.

The **VERSION** attribute is used to distinguish future versions of the language. Currently the value needs to be “1.0”. Weavers should check this against their own version.

PLATFORM – The application's platform information

<i>Syntax</i>	<PLATFORM />
<i>Attribute specifications</i>	• LANGUAGE= <i>string</i>
<i>Contents</i>	<i>No contents.</i>
<i>Contained in</i>	CONNECT

PLATFORM defines the environment of the application. All of the other platform-specific parts of the Connect file must conform to this.

Currently, **PLATFORM** has no contents, and only one attribute: **LANGUAGE**. The value can be either “C++” or “Java”. In the future, Connect could support other languages, or compilers that need special care. Support for this shall all be in this tag.

CLASSES – Container of class tags

<i>Syntax</i>	<CLASSES> ... </CLASSES>
<i>Attribute specifications</i>	<i>No attributes.</i>
<i>Contents</i>	Zero or more CLASS tags
<i>Contained in</i>	CONNECT

CLASSES is a container for **CLASS** tags. It is used to group **CLASS** tags. Grouping **CLASS** tags has no immediate use, instead, it can be used to logically separate tags for different subsystems in the application. Each subsystem's classes could be placed in a different **CLASSES** section.

CLASS – Definition for a class' information

<i>Syntax</i>	<CLASS> ... </CLASS>
<i>Attribute specifications</i>	<ul style="list-style-type: none">• NAME=<i>classname</i>• FILENAME=<i>filename</i>
<i>Contents</i>	Zero or more SIGNAL tags
<i>Contained in</i>	CLASSES

Every class in the application that has it's own signals needs to have a **CLASS** tag with it's fully qualified name. Signal proxies do not need to have class tags, unless they have signals of their own.

The **NAME** attribute is the fully qualified name of the class, including any scopes (for internal classes), paths (for Java classes) or namespace (in C++). The format of **NAME** depends on the contents of the **PLATFORM** tag.

FILENAME is a path to the file (either absolute or relative) which contains the given class and which the weaver will modify, if necessary. For C++, this must be the header in which the class is declared, for Java, the class' source file.

It should be noted that a class that will only act as a receiver for signals doesn't need to have it's CLASS tag, as slots are created at runtime.

SIGNAL – A class signal's signature

<i>Syntax</i>	<SIGNAL />
<i>Attribute specifications</i>	<ul style="list-style-type: none">• NAME=<i>methodname</i>• OUTPUT=<i>parameters</i> (optional)
<i>Contents</i>	<i>No contents.</i>
<i>Contained in</i>	CLASS

Each signal that a particular class has needs to have it's own **SIGNAL** attribute.

NAME and **OUTPUT** define the signal's signature. If the output tag is missing, the signal is void (takes no parameters). For example, to define a signal, temperatureChanged(**double**,**bool**) the tag would be:

```
<SIGNAL NAME="temperatureChanged" OUTPUT="double,bool" />
```

Many classes can have signals with the same name. Signals can be overloaded if the platform supports method overloading (both C++ and Java does). Signals are inherited by subclasses like methods.

Note: In C++, the '&' (ampersand) is used for object references. XML requires ampersand to be encoded as '&'. The XML parser will report it as a fatal error if we forget this.

ADAPTERS – Container of adapter tags

<i>Syntax</i>	<ADAPTERS> ... </ADAPTERS>
<i>Attribute specifications</i>	No attributes.
<i>Contents</i>	Zero or more ADAPTER tags
<i>Contained in</i>	CONNECT

ADAPTERS is a container for **ADAPTER** tags. It is used to group **ADAPTER** tags. As with **CLASSES**, grouping has no effect other than the logical structuring of adapters.

ADAPTER – Definition of a signal adapter

<i>Syntax</i>	<ADAPTER />
<i>Attribute specifications</i>	<ul style="list-style-type: none">• NAME=<i>classname</i>• INPUT=<i>parameters</i>• OUTPUT=<i>parameters</i>
<i>Contents</i>	No contents.
<i>Contained in</i>	ADAPTERS

Every adapter in the application needs to be defined with an **ADAPTER** tag.

NAME is the name of the adapter's class.

INPUT is the list of parameters that the adapter's slot takes. **OUTPUT** is the parameters of the output signal. If missing **INPUT** or **OUTPUT** attributes imply void input or output, respectively. (Don't forget to write ampersand's as '&';!)

For example, the adapter example in chapter 2.1.4. would have the following tag:

```
<ADAPTER NAME="CADPointToGUI" INPUT="double,double"  
OUTPUT="Point" />
```

The adapter's body (the code which converts the values) is defined elsewhere, depending on the platform. See the Implementation chapter.

TRANSITIONS – Container of transition tags

<i>Syntax</i>	<TRANSITION> ... </TRANSITION>
<i>Attribute specifications</i>	No attributes.
<i>Contents</i>	Zero or more TRANSITION tags
<i>Contained in</i>	CONNECT

TRANSITIONS is a container for **TRANSITION** tags. It is used to group **TRANSITION** tags. The grouping is only logical, and has no other effect.

TRANSITION – Definition of a transition

<i>Syntax</i>	<TRANSITION> ... </TRANSITION>
<i>Attribute specifications</i>	• NAME= <i>classname</i>
<i>Contents</i>	Two or more SLOT tags
<i>Contained in</i>	TRANSITIONS

Every transition in the application needs to be defined with a **TRANSITION** tag.

NAME is the name of the transition's class.

The input slots need to be defined as **SLOT** tags (similar to **SIGNAL** tags, but have **INPUT** instead of **OUTPUT**. Don't forget to replace ampersands with '&');).

The example transition in Chapter 2.1.5 would have the following tag:

```
<TRANSITION NAME="tripleTransition">
  <SLOT NAME="slot1" INPUT="double,bool" />
  <SLOT NAME="slot2"/>
  <SLOT NAME="slot3" INPUT="Line&,int" />
</TRANSITION>
```

Following is an example Connect file:

```
<?xml version="1.0"?>
<!DOCTYPE CONNECT>
<!-- This is an example Connect file, handling the Thermometer
example from chapter 1.6 -->
<CONNECT VERSION="1.0">

  <!-- The first tag of CONNECT must be PLATFORM. -->
  <PLATFORM LANGUAGE="C++" /> <!-- This is a C++ application. -->

  <!-- The following parts deal with connecting the Domain's
CpuGuard and Thermometer classes to each other and to the GUI's
widgets. -->

  <CLASSES>

    <!-- The Thermometer class emits signals to notify others
whenever the temperature changes. -->
    <CLASS NAME="Domain::Thermometer">
      <!-- Simple void signal, the receiver will probably
poll the new value, or make others do it -->
      <SIGNAL NAME="temperatureChanged" />
      <!-- This signal passes the new temperature value on,
in degrees Celsius -->
      <SIGNAL NAME="temperatureChanged" OUTPUT="double" />
      <!-- This signal passes the new temperature value on,
in degrees Fahrenheit -->
      <SIGNAL NAME="temperatureChangedF" OUTPUT="double" />
    </CLASS>

    <!-- CpuGuard has an int alertLevel property, which depends
on the temperature. -->
    <CLASS NAME="Domain::CpuGuard">
```

```

        <!-- alertLevel change. One of the other alertLevel
        signals is also emitted. -->
        <SIGNAL NAME="alertLevelChanged" OUTPUT="int" />
        <!-- alertLevel==0 means OK, this signal emitted, when
        alert level changes to 0 -->
        <SIGNAL NAME="alertLevelOK" />
        <!-- alertLevel==1 means the temperature is too high,
        but not dangerously so -->
        <SIGNAL NAME="alertLevelWarning" />
        <!-- alertLevel==2 means the temperature is critical-->
        <SIGNAL NAME="alertLevelCritical" />
    </CLASS>

</CLASSES>

<ADAPTERS>

    <!-- The temperature is displayed in a text label. Need to
    convert the double to unicode string. Since this is a common
    function, we place the adapter in the global namespace. -->
    <ADAPTER NAME="::DoubleToString" INPUT="double"
        OUTPUT="String"/>

    <!-- An icon starts flashing when the CpuGuard's alertLevel
    reaches 2.
    The GUI icon class has a setFlashing(bool) method, so the
    integer needs to be adapted.
    This adapter return true if the parameter integer is 2. -->
    <ADAPTER NAME="GUI::AlertLevelToCritical" INPUT="int"
        OUTPUT="bool" />
</ADAPTERS>

<TRANSITIONS>
    <!-- .... -->
</TRANSITIONS>

    <!-- More CLASSES and other tags could follow, dealing with other
    parts of the application. -->

</CONNECT>

```

2.3. Support Classes

The Connect language describes parts of the signal/slot mechanism. This description is used for generating some of the code.

We also require mechanisms to:

- Define slots,
- Connect and disconnect signals,
- Create and destroy transitions,
- Define and undefine signal proxies.

These tasks need to be done at **runtime**, and for these we need some support classes to handle them. The classes themselves depend on the platform in a number of ways.

- Is the language strongly typed?
- How does the language support pointers to functions or methods?
- How do we call a method of an object, if we're not sure at compile time *when* and *where* we need to make that call?

C++ is a strongly typed language. This is a Good Thing, because C++ compilers can catch a lot of programming errors. The drawback is that as soon as we try to do something like a slot call, we need to jump through a number of hoops before it compiles. In exchange, we get very good compile-time type checking. C++ has templates and function pointers to help us out.

Java is also strongly typed, but it has a number of facilities, such as meta-objects, which make our life easier. However, the designers of Java have eliminated the function pointer, replacing it with interfaces and reflective programming.

With both languages, Connect has a central object, conveniently named “Connect”, which keeps track of connections and signal proxies. This helps isolate the communication aspect from the rest of the code, resulting in nice, clean objects, almost like the happy standalone example in Chapter 1.3. Signal proxies and receiver objects don't need to be modified at all to be used with Connect. Also, it is this architecture which allows for signal proxies.

3. Implementation of Connect

Implementing Connect for a given language means creating a **weaver**, which modifies the code of the original software, and adds the code for communication. The weaver inserts code for signals and support classes, producing the *final* source code for the application.

I have found that mature code transformation tools are available for Java, however, C++ is much harder to transform. Luckily, inserting the code for signals is not very difficult for either language, even parentheses matching will work. In the following chapter, I will describe the inserted code and the weaving process for both languages.

In implementing Connect, I will try to make **as little modification to objects as possible**. Objects which have signals will have to be modified *per se*: code for the signals need to be inserted. Also, the developer is responsible for emitting the signals whenever it is necessary. I have considered *automatically emitted (implicit) signals*, for example, whenever the internal state (descriptive attributes) of an object changes. Finally I decided to use explicit signals instead: implementing implicit signals is needlessly difficult and modifies the code too much.

Classes which don't have their own signals **don't need to be modified at all**. For example, any object can be a signal proxy, and any object can be a receiver of a signal. This is a must; we consider some of the receiver classes to be immutable; for example in most cases we cannot modify the GUI widget set, sometimes we may not even use subclassing due to final objects or private inheritances which provide a completely stripped interface (I had some very unpleasant experiences with that).

3.1. Implementation in C++

Signals and Slots can be implemented in C++ in a number of ways. We need a mechanism to define signals, slots and pass arguments.

TrollTech's **Qt widget set** uses a metaobject compiler, which modifies every object which has signals and/or slots. This means that a slot cannot be just any function or method. The metaobject compiler inserts code which serializes the parameters in the signals and deserializes them in the slot. This allows for automatic type conversions. Qt's signals are identified by strings, which means that no compile-time type checking can be performed. Qt gives warnings at runtime when incompatible signals and slots are connected. Objects themselves store the list of their receivers. There are no adapters or signal proxies.

Another player, **LibsigC++**, was originally created to replace callbacks for gtk--, a C++ wrapper for another widget set. LibsigC++ allows signals and slots to be defined at runtime and allows strong type checking at compile time. The signals store the list of their receivers. LibsigC++ has many advanced features, but lacks signal proxies. I have based some of my implementation code on LibsigC++.

Some application frameworks use proprietary language extensions to achieve functionalities similar to signals and slots. However, these solutions restrict the software to a single compiler. In the brave new world of software this is not acceptable.

Connect's implementation is different from both Qt's and LibsigC++'s, taking advantage of a weaver. The weaver reads the Connect file, and transforms the source files based on its information. We use **declared signals**, like Qt, **undeclared slots**, like LibsigC++, and a central object, Connect, to maintain the connection lists, and signal proxies.

In implementing Connect, I have put great emphasis on type safety. All of the checking is done at compile time, so there should be no unpleasant surprises at runtime.

3.1.1. Creating slots

Ideally, a slot is a lightweight object, which has a method, call(), that takes just the right parameters. Calling this method will then invoke a function or an object's method.

We could generate code for all possible slots – every method in every class, which has a signature that matches, or can be adapted to one of the signals. While this is simple enough, the amount of generated code is far too much.

Our other option is using **templates**. We want to end up with a template class, which can be instantiated with any signal signature, and called with a given set of parameters. Since C++ doesn't support variable number of parameters for a template, we need a different template for different number of parameters. This means we will need to create Slot0, Slot1, Slot2, ... templates to support slots with 0, 1, 2, ... arguments respectively. We'll go up to 7 arguments, they say more than 7 arguments are too many anyway. Starting bottom up, we need a structure, **SlotData** which holds information required to call the right function:

```
struct SlotData
{
    /* Callback function. This is a static function in one of the
    Slot# classes, to handle the data. This is a replacement for a
    virtual function. The signature is a dummy signature, the Slot
    will cast it to the right form. This is safe, because Slot
    contains the structure anyway. */
    void* (*callback)(void*);

    /* Adapter function. This is a static function which calls the
    callback with the parameters modified. */
    void (*adapter)(void*);

    struct O;
    struct C1
    {
        // Simple function with no object
        void* (*f1)(void*);          // The function
        void* dummy;                 // A placeholder
    };
    struct C2
    {
        // Method of an object
        void (O::*v)(void);          // The method
        O* o;                        // The object
    };

    // Object pointer or function pointer
    union { C1 a1; C2 a2; };
};
```

SlotData has two possible casts, SlotDataFunc and SlotDataObj:

```
template <class C, class F> struct SlotDataFunc
{
    C callback;
    void (*adapter)(void*);
    F func;
};

template <class C, class O, class F> struct SlotDataObj
{
    C callback;
    void (*adapter)(void*);
    F func;
    O *obj;
};
```

Now suppose we want a slot with two parameters.

```
template <class PAR1, class PAR2> class Slot2 : public SlotData
{
public:
    typedef void (*Callback)(void*,PAR1,PAR2);
    typedef void (*Adapter)(void*,void*(*calb)(void*),PAR1,PAR2);

    Slot2( SlotData _data ) : SlotData( _data ) {};

    inline void call( PAR1 p1, PAR2 p2 )
    {
        if (adapter)
        {
            ((Adapter)(adapter))(this,callback,p1,p2);
        }
        else
        {
            ((Callback)(callback))((void*)this,p1,p2);
        }
    }
};
```

We still require the Callback for the slot's method or the function. Let's consider the method first. We need another template to produce the callback function, and also a factory to create the right Slot2 object. Presenting, the **ObjectSlot2** class, which has nothing but two static functions:

```
template <class T, class RET, class PAR1, class PAR2> struct
ObjectSlot2
{
    typedef RET (T::*Method)(PAR1, PAR2);
    typedef void (*Callback) (void*, PAR1, PAR2);
    typedef SlotDataObj<Callback,T,Method> SlotData_T;

    /* Instantiate a callback for this particular class with two
    parameters. The SlotData structure is passed to it by
    Slot2::call() as the first argument. */
    static void callback( void* d, PAR1 p1, PAR2 p2 )
    {
        SlotData_T* data = (SlotData_T*) d;
```

```

        ((data->obj)->*(data->func))( p1, p2 );
    }

    // Instantiate a factory method for this particular class with
    // two parameters
    static Slot2<PAR1,PAR2> create( T* receiver, Method method )
    {
        SlotData tmp;
        SlotData_T& data = reinterpret_cast<SlotData_T&>(tmp);
        data.callback = &callback;
        data.obj = receiver;
        data.func = method;
        return Slot2<PAR1,PAR2>(tmp);
    }
};

```

A similar class, **FunctionSlot2**, can be defined for function slots.

There should also be a factory function, **slot()**, which takes a function or object/method pair, and returns the right slot, to make the the life of the programmer easier. The slot() function for ObjectSlot2 would look like this:

```

template <class T, class T2, class RET, class PAR1, class PAR2>
inline Slot2<PAR1,PAR2> slot(T* receiver,
    RET (T2::*method)(PAR1,PAR2) )
{
    return ObjectSlot2<T,RET,PAR1,PAR2>::create(receiver,method);
}

```

(Note that T2 is there to support inherited methods. It must be a method of receiver, otherwise ObjectSlot2 instantiation will fail. This is detected at compile time.)

For the programmer who will use our slots, this is a very simple matter; he merely calls slot() and receives a Slot2 object of the right type. In spite of all the casts, the whole method is completely typesafe.

3.1.2. Adapters

Adapters are easily added to the slots described in the previous chapter. The SlotData structure has an adapter field, which is NULL by default. If we assign a value to this field, the Slot?::call() function calls the adapter instead, passing the parameters and the callback to it.

The adapter then calculates the parameters for the real callback, and calls it.

In effect, the slot will appear to have changed its type, which is what we wanted to achieve with adapters.

For example, take the CADPointToGUI adapter from chapter 2.1.4. That weaver would insert the following code:

```

void CADPointToGui( void* data, void (*callback)(void*, Point), double
p1, double p2 )
{
    Point point( int(p1), int(p2) );    //!

    (*callback)( data, point );
}

```

```

template <class T, class T2, class RET>
inline Slot2<double,double> slotWithCADPointToGui( T* receiver, RET
(T2::*method)(Point) )
{
    Slot2<Point> _slot = slot( receiver, method );
    _slot.adapter = (void (*)(void*)) &CADPointToGui;
    return reinterpret_cast< Slot2<double,double> >(_slot);
}

```

```

template <class RET>
inline Slot2<double,double>
slotWithCADPointToGui(RET(*function)(Point))
{
    Slot2<Point> _slot = slot( function );
    _slot.adapter = (void (*)(void*)) &CADPointToGui;
    return reinterpret_cast< Slot2<double,double> >(_slot);
}

```

All of this code is automatically generated by the weaver, except for the line marked with '///'. That piece of code makes the conversion.

This kind of adapter is very lightweight in terms of memory usage.

Unfortunately, not every conversion can be done with a simple static function; for some conversions, a “stateful” adapter is required. For example, suppose a stream is reading Russian text encoded in the Koi8-r format. Most of our application uses Unicode to store text. To automate the conversion from Koi-8 to Unicode, we could add an adapter to transform the text as it is transferred from the stream to some other classes with signals and slot. The problem is, that some characters have two bytes, and a character could be split between two chunks of data. To handle this, we would need a stateful adapter, one that could remember the context, or at least the last character.

Our adapters can be hacked into stateful adapters by using the data* argument of the adapter function as an index to a more complex data structure.

3.1.3. Signals

Signals are methods of classes. Emitting a signal is simply calling the signal method. The signal methods themselves are inserted into the class' source by the weaver. The signal method will acquire from Connect the list of slots it needs to call, and then proceed to call them.

Every signal function is woven into the class by the weaver based on the Connect file.

An example signal function would look like this (pseudo-code):

```

void Thermometer::temperatureChanged( double p1 )
{
    // The connection list for signal #2 of this object
    ConnectionList list = Connect->getConnectionList( this, 2 );
    Slot1<double> slot;
    ConnectionListIterator it;

    // Iterate the list, call every slot
    for ( it=list.begin(); it!=list.end(); ++it )

```

```

    {
        slot = *it;
        slot.call( p1 );
    }
}

```

Actually this code is not in the class instead. It's a method of Connect, `invoke_temperatureChanged(double)`.

3.1.4. Connect

Connect is a singleton object responsible for making and deleting connections and signal proxies.

Connect has a connect and disconnect methods for every kind of signal signature. These functions only take slots of the right type, which allows for compile time type checking. Every connect and disconnect method is woven into Connect based on the Connect file. Continuing the example in chapter 2.2, Connect would have the following public methods:

```

// connect_* functions
bool connect_temperatureChanged( void* source, Slot0 target );
bool connect_temperatureChanged( void* source, Slot1<double> target );
bool connect_temperatureChangedF( void* source, Slot1<double> target
);
[...]

// disconnect_* functions
bool disconnect_temperatureChanged( void* source, Slot0 target );
[...]

// invoke_* functions
bool invoke_temperatureChanged( void* source, Slot0 target );
[...]

// Disconnects every signal from source object
void disconnectSource( void* source );
// Disconnects every signal to the target object
void disconnectTarget( void* target );
[...]

// Makes proxy a signal proxy for source.
void createSignalProxy( void* source, void* proxy );
// proxy will no longer be a signal proxy for source.
void destroySignalProxy( void* source, void* proxy );
// Returns the list of signals connected a given signal of source
ConnectionList getConnectionList( void* source, int signalIndex );

```

`disconnectSource` and `disconnectTarget` are useful to call from destructors, they will remove all connections from/to the given objects, as well as any signal proxy relationships.

`getConnectionList` is used by signal functions. It returns a collection of every signal which is connected to the source's signal, either directly or indirectly. Every kind of signal is assigned a unique index by which it can be identified by Connect.

Internally, Connect uses a `Connect_Base` class which hides the gory details of Connect's functionality.

3.1.5. Transitions

The implementation of transitions is very straightforward: their slots store incoming signal's parameters, and set a flag. When every slot's flag is set, the `activated()` signal is emitted, with the stored parameters.

3.1.6. Weaving Steps

In processing the Connect file, the weaver needs to do the following steps

1. Create Transitions.
2. Generate code for Adapters, and merge it with previously created adapters, to preserve the user's modifications.
3. Insert `connect_*` and `disconnect_*` and `invoke_*` into Connect's class.
4. Insert the signals' code into every object which has signals.

Lacking good code transformation tools, the C++ weaver is a PERL script which inserts code based on parentheses matching. PERL is a scripting language well known in the UNIX world, and liked for its powerful text processing capabilities and huge set of modules. PERL has an incarnation under Win32 (called ActivePerl), and Mac OS. I have tried ActivePerl myself, and found it equally capable as the UNIX version.

The weaver (**CPPWeaver.pl**) takes up to three parameters. The first parameter is the name of the Connect file, the second is the filename into which the Connect object will be written (.h and .cpp are appended), the third is the filename for the adapters. The first parameter is required, the others are optional, and default to 'connect' and 'adapters.h' respectively.

The weaver parses the XML file, and modifies the source files if necessary. Care is taken not to modify files unless necessary.

An example C++ project using Connect is included with this text on the enclosed floppy disc.

3.2. Implementation in Java

I will not go into great detail on the Java implementation of Connect. Java is similar to C++ in many respects, the only great difference is how slots are handled.

3.2.1. Creating slots

As I have shown, slots are essentially callbacks. For a variety of reasons, the Java language has eliminated function pointers. Most of the use of callbacks is replaced by interfaces. We don't have templates either, so we are left with the Reflect API [Nie97].

The `comp.lang.reflect` package has a `Method` class, which holds the key.

Our unified slot class will look something like this:

```
import java.lang.reflect.*;
```

```

public class Slot
{
    Method method;
    Method adapter;
    Object receiver;

    // Class method constructors

    public Slot( Method m ) {...}
    public Slot( Class c, String signature ) {...}
    public Slot( String classname, String signature ) {...}

    // Instance method constructors

    public Slot( Object o, Method m ) {...}
    public Slot( Object o, String signature ) {...}

    // Setting an adapter

    public void setAdapter( Method a ) {...}
    public void setAdapter( String adapterName ) {...}

    // Call method

    public void call( Object[] args )
    {
        method.invoke( receiver, args );
    }
}

```

Slots can be instantiated either by a Method or by their signature. For example:

```
Slot s( guard, "void newTemperature( double )" );
```

would create a Slot object which would call the given method of the object guard. Unfortunately we lose compile-time type checking with this construct. If I had better knowledge of the Java language I may find a better solution to this.

3.2.2. Adapters

Adapters in the Java version of Connect are static methods of one class, called Adapters. This class is created and maintained by the weaver. As with the C++ version, the programmer needs to write the function bodies.

Slot provides an interface to set an adapter either by name or by Method.

3.2.3. Signals

Signals are handled same as the C++ version. The signals are methods of the classes. The signal method creates the args[] array required for the slots, gets the list of slots from Connect and calls() them.

3.2.4. Connect

Connect's interface is very similar to the C++ version.

Instead of `void*`, Java uses `Object` parameters (obviously). The `connect_*` methods implement runtime type checking, giving warnings on incorrect connections.

3.2.5. Weaving steps

Weaving C++ was problematic because there were no software packages for transforming code. Weaving Java is problematic because there are too many options, and it's hard to choose.

RECORDER

RECORDER is a Java framework for source code metaprogramming aimed to deliver a sophisticated infrastructure for many kinds of Java analysis and transformation tools [REC01]. RECORDER can analyze and transform Java code.

BOXOLOGY

The BOXOLOGY framework defines a composition model and operators for invasive software composition.

COMPOST

COMPOST stands for Software Composition System, allows to engineer, re-engineer and evolve software by program transformations [COM01]. It is best adapted at composing applications from components, adapting them to each other by means of program transformations.

AspectJ

AspectJ, a feature-complete AOP solution for Java has a compiler, a debugger, and IDE extensions to provide a complete infrastructure for aspect-oriented programming, by Xerox Co.. [ASPJ98] AspectJ can modularize a lot of crosscutting concerns. AspectJ is a *language extension* to Java, designed to be simple and practical. [XER01]

Inject/J

Inject/J is a powerful scripting language that can perform source code transformations. The language provides statements for navigating and transforming the Abstract Syntax Tree, a tree representation of the Java source code [Kut99]. Inject/J actually uses RECORDER to implement its operations.

Considering the above, I have chosen Inject/J, as it has the best features for the Java Weaver for Connect. The weaver itself (**JavaWeaver.pl**) produces and Inject/J script and calls Inject/J to take care of the rest. It's funny, how a *Connect* file is read by a *PERL* script to produce an *Inject/J* script to modify *Java* code.

The Java weaver takes only one parameter, the filename of the Connect file. Some environment variables (INJECTJ_HOME and CLASSPATH) should also be set. Other than the extra scripting step, the weaving process is same as for C++.

4. Mixing Connect with Design Patterns

Design patterns [GAM95] are simple, insightful solutions to specific object-oriented design problems. Implementing design patterns may take some extra effort, but that effort pays off by increasing the flexibility and reusability of the software.

Using design pattern in *designing* a software is different from *implementing* them in the actual code. Implementing design patterns requires no special skill, in fact, it be a lot of mechanical work, involving some cutting and pasting of code. Connect can help implement some patterns that relate to references between objects. Connect can require modification in implementing of others. In this chapter, I will discuss using Connect with design patterns.

All of the patterns in this chapter were taken from [GAM95]. The class diagrams are modified and show how Connect interacts with these patterns.

4.1. Structural Patterns

Structural design patterns help form larger units of classes.

Connect isn't concerned with composition, and therefore offers little help in implementing these patterns (although some patterns, can be replaced with creative usage of Connect's features). Besides the key feature, connecting signals and slots, there are some other features that help.

- Slot objects can be used to export a callback without knowledge of the target object's class, interface or memory location.
- Signal proxies can help decouple abstractions and implementation
- Transitions can help with synchronization.

I will describe ways how Connect can be adapted to various structural patterns. Usually we have a structure composed of two or more classes, and we need to deal with two issues:

1. Forwarding signals coming from the outside world from the 'outer' classes to the 'inner' (hidden) classes. Most structural patterns hide their internal classes by outer classes. However, the outside world can only send signals to the outer classes, hence the need for signal forwarding.
2. Forwarding outgoing signals from the 'inner' classes to the outside world.

Our design needs to make the signals as transparent as the entire design is. Now is the time when we'll see how important signal proxies are.

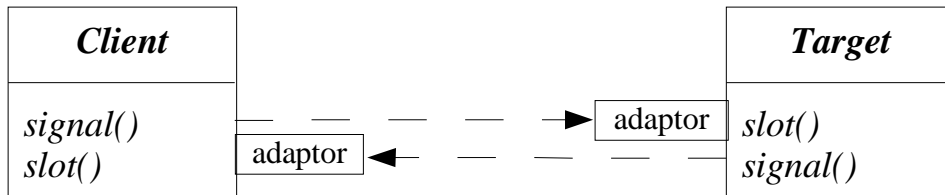
4.1.1. Adaptor

First, it is important to note that the Adaptor design pattern is not the same as the adapters in Connect (chapter 2.1.4). The Adaptor design pattern changes the interface of a class, while our adapter changes the calling parameters of a slot. For this reason, I will use the different spelling and the capital letter to distinguish them.

Adaptor replaces the interface of a class, *Adaptee*, with the interface *Target*. The *Adaptor* class either uses multiple inheritance (inherits both *Target* and *Adaptee*), or object composition (contains an *Adaptee* object, and forwards requests to it).

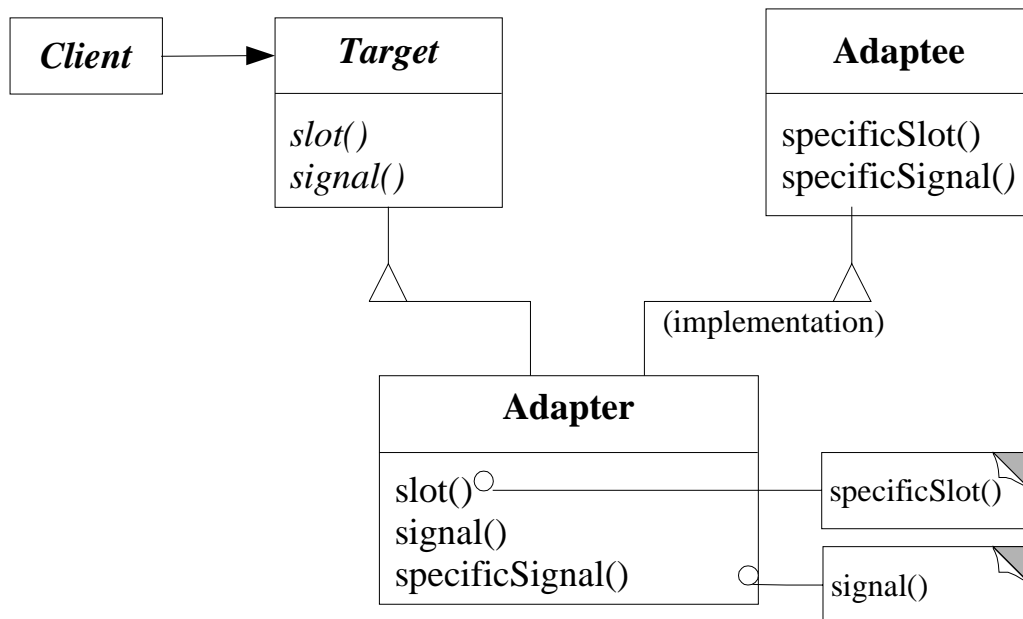
What Connect helps implementing

Connect can be used to *replace* the Adaptor pattern, if all of the requests to *Adaptee* can be directly replaced with signals. In this case we can define a proper adapter for every signal, and we are done.



If the requests cannot be handled in this way, then Connect doesn't help to implement Adaptor.

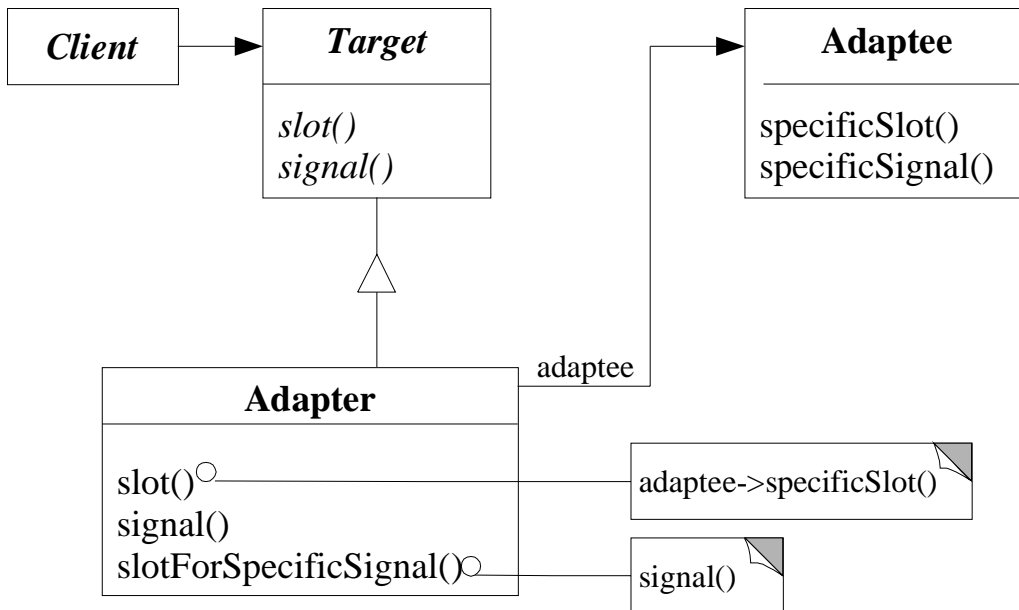
How Connect can be used with this pattern



Let's consider the case of a class adapter first.

1. *Incoming signals*: For every slot method, the Adapter needs to call the Adaptee's `specificSlot()` method (this is the same as with any normal method).
2. *Outgoing signals*: As for signals in the Adaptee, we need to *override* the (generated) `specificSignal` method, and call `signal()` instead.

This is the case of of an object adapter.



1. *Incoming signals*: Slots are handled similarly to regular method adaptations.
2. *Outgoing signals*: For every `specificSignal()`, the Adapter needs to define a slot, `slotForSpecificSignal()`, which connects to the `adaptee`'s `specificSignal()`, makes the appropriate transformation, and calls `signal()`.

This makes signals and slots transparent to clients.

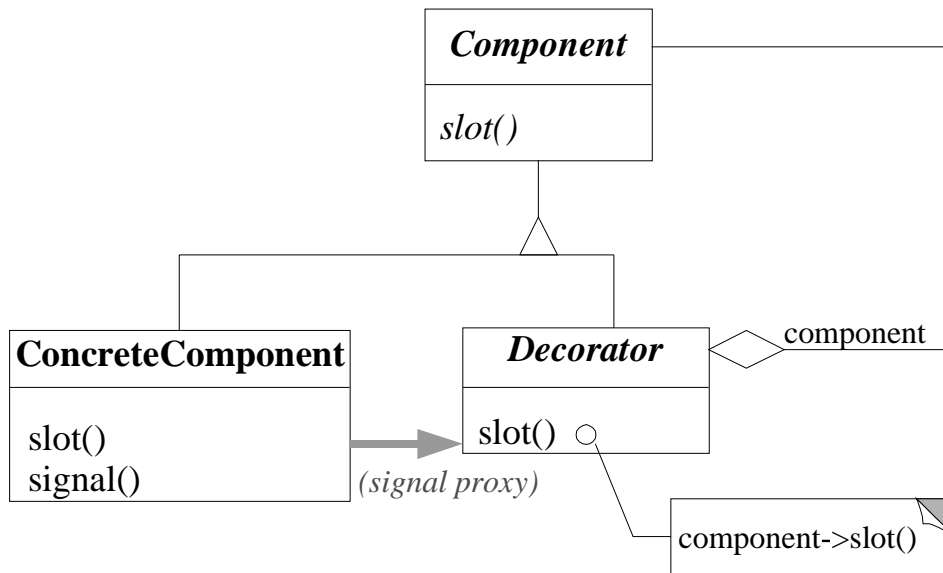
4.1.2. Decorator

Decorators are used to dynamically add responsibilities to objects. This is achieved by enclosing a component in another object (the Decorator) that adds additional functionalities. The Decorator is transparent (it has the same interface as the Component) and forwards requests to the Component it decorates, and may perform additional actions.

What Connect helps implementing

Connect offers no help implementing the Decorator pattern.

How Connect can be used with this pattern



1. *Incoming signals:* The Component's and its subclasses' slots are simply called by the Decorator's slots.
2. *Outgoing signals:* The Component's signals, on the other hand, are proxied by the Decorator.

This makes signals and slots transparent to clients.

4.1.3. Bridge

The **bridge pattern** decouples an abstraction from its implementation.

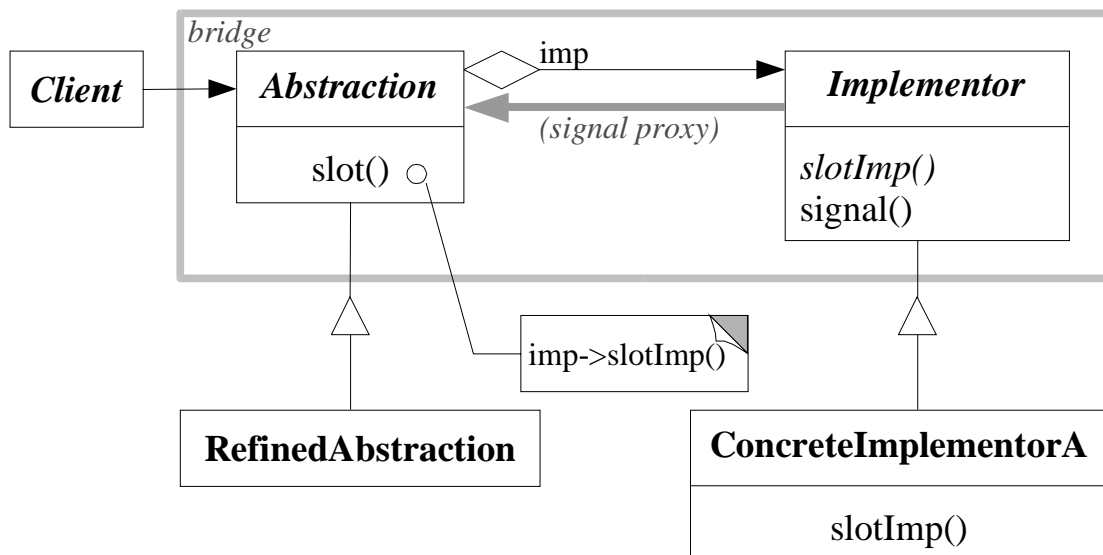
The **Abstraction** is an interface class, and usually has a single attribute, a pointer to the **Implementor** class. Both the Abstraction and the Implementor can be extensible by subclassing. Bridge is common in C++ for hiding the details of the implementation from clients.

What Connect helps implementing

Connect offers no help implementing the Bridge pattern.

The Bridge pattern can, however, be replaced by signals and slot, if you connect the Abstraction's every request to the Implementor's slots. The Implementor's signals can in turn be proxied by the Abstraction. In fact, you can use a different Implementor for every signal.

How Connect can be used with this pattern



1. *Incoming signals:* The Abstraction can call the Implementors's slots.
2. *Outgoing signals:* It is clear that the Abstraction class will have no relevant signals. The Implementor, and it's subclasses may provide useful signals, but they are hidden by the Abstraction and invisible from outside. Signal proxies help: if you make the Abstraction's instances signal proxies for Implementors, signals from Implementor subclasses pass onto the Clients transparently.

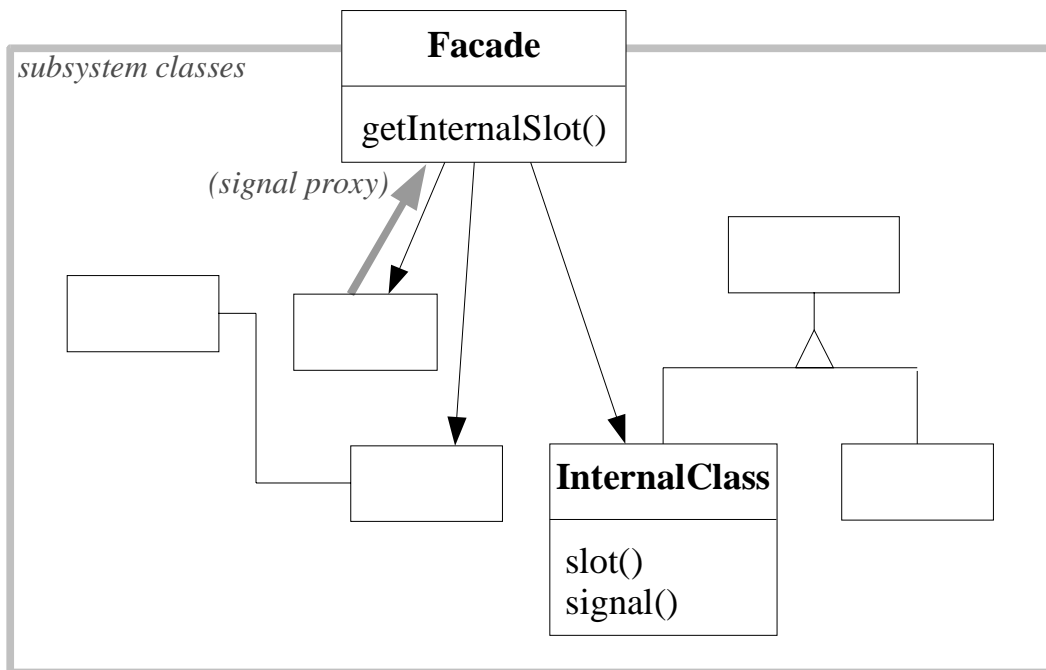
4.1.4. Facade

The **Facade** design pattern provides a simple, unified interface to a complex subsystem. The Facade is a simple default view, suitable for most Clients. Changes to the subsystem don't affect the Facade. Only clients needing more powerful, lower-level interfaces need to look beyond the Facade. The Facade eliminates a lot of unnecessary dependencies between Clients and the internal classes of the subsystem.

What Connect helps implementing

Connect helps the Facade indirectly. The Facade can provide signals that Clients can connect to, or can export Slots of internal objects. Since the Slots don't actually contain class information, this design allows for more powerful interfaces without impacting the software with unnecessary dependencies.

How Connect can be used with this pattern



Facade can be combined with Connect's facilities in a number of ways.

- The Facade can act as a *signal proxy for some internal objects*. This is not as powerful as in the case of other patterns, because many internal objects may have the same signals, and Client object may only want to connect to one of them. However, themselves
- The Facade can provide slots to *forward and filter internal objects' signals*.
- The Facade can *forward signals from Clients* to internal objects.
- The Facade can *export slots of internal objects*. This involves creating Slot objects and handing them out to Clients.

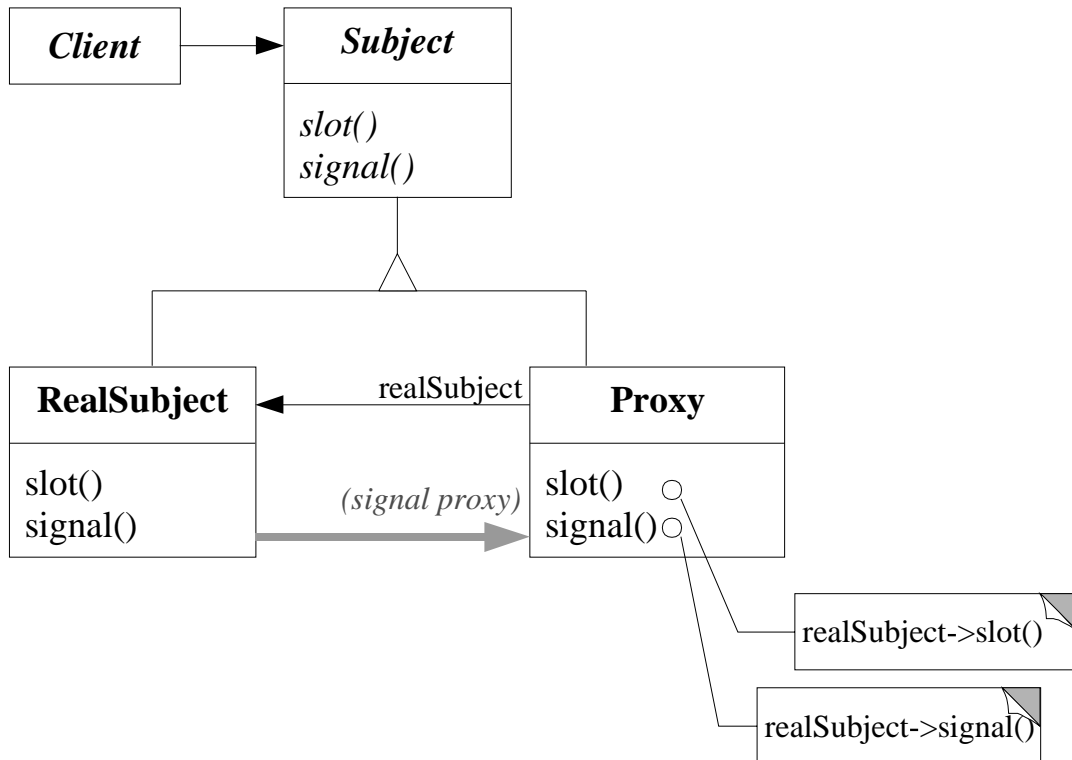
4.1.5. Proxy

A Proxy is a placeholder of another object. The Proxy is in effect a smart reference.

What Connect helps implementing

Connect offers no help implementing the Proxy pattern.

How Connect can be used with this pattern



1. *Incoming signals:* Proxy calls the RealSubject's slot.
2. *Outgoing signals:* The abstract signal() function of Subject is implemented in RealSubject by Connect. The Proxy's signal() function calls RealSubject's signal(). Since Clients access Proxy instead of RealSubject, Proxy is made a signal proxy of RealSubject.

An alternate method is not to define signal() in Subject. RealSubject will still have the signal, and Proxy will copy it as before.

4.2. Behavioral Patterns

Behavioral patterns deal with algorithms and sharing responsibilities between objects. Connect is helpful with patterns that describe communication and collaboration between objects. There are two behavioral patterns, the Mediator and Observer, which can be directly implemented with Connect.

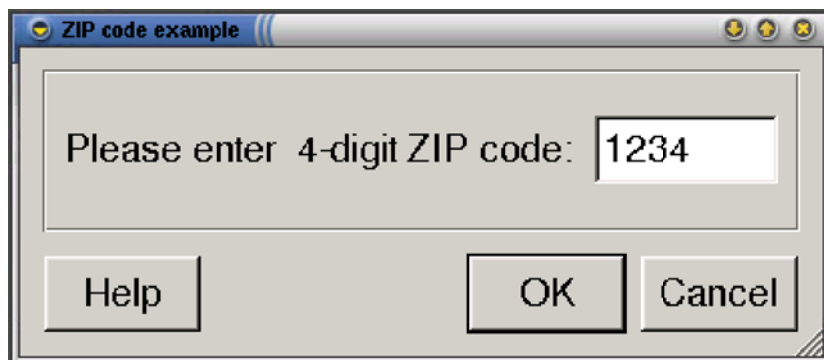
4.2.1. Mediator

The **Mediator** design pattern encapsulates communication between objects (Colleagues) in a separate object. Mediator achieves loose coupling of these objects by keeping them from referring to each other explicitly.

What Connect helps implementing

Connect can be used to completely replace mediators. In fact, Connect *is* a generic mediator, promoting interaction while preserving loose coupling.

As an example, suppose we have a dialog where the user must enter a ZIP code. The ZIP code is (in this context) a number with exactly 4 digits. The dialog has an “OK” button. To help the user, we disable the OK button if the text is not a 4-digit number.

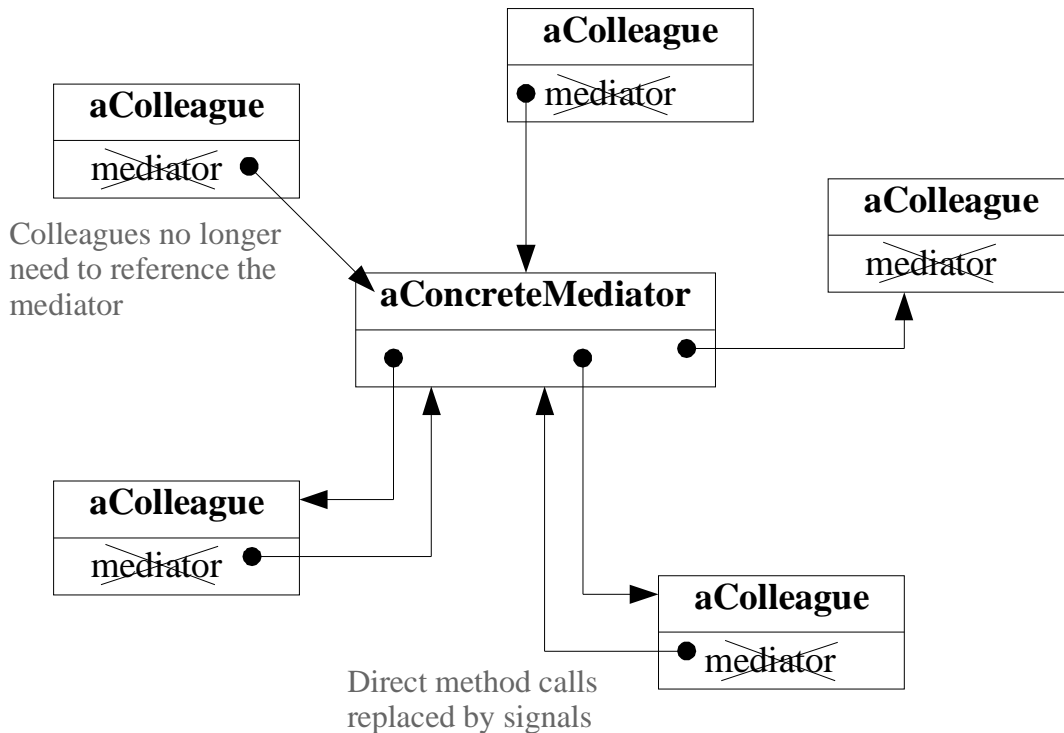


Normally, we could use a Mediator class which would be notified by the Editor object whenever it changes (this could require modification to the Editor, either by subclassing or by Decorating). The Mediator would then enable or disable the OK button. We had to introduce at least two classes!

With connect, our life is much easier. The Editor shall have a signal, **textChanged(String)**. The Button shall have a method, **setEnabled(bool)**. We define an adapter, **ZIPToBool**; this adapter converts String to bool, and returns true if the String is a valid ZIP code (four digits). Then we simply connect the **textChanged(String)** signal to the **setEnabled(bool)** signal with **ZIPToBool**, and we're done. Connect will take care of the rest.

The Button, in turn, would have a **clicked()** signal, which we could connect to the dialog's **accept()** slot. All the dialog's functionality, with a few generic signals and slots.

How Connect can be used with this pattern



More complex cases cannot be solved with adapters and transitions, and still require Mediators. The Mediator's life, however, can be made easier with Connect. Replacing callbacks and direct method calls with signals and slots can both take on parts of the Mediator's responsibilities. Also, this allows for multiple Mediators to operate on some Colleagues, without the Colleagues having to keep references of all of them.

Moreover, using signals and slots in Mediators spares us from adapting, subclassing or otherwise modifying generic objects. The object no longer need to reference the mediator either. In the ZIP code example, the same Editor and Button could be used in a number of dialogs, without having to modify them to accommodate the dialog's specific Mediator class. Many design patterns, when applied, tend to increase the number of classes; implementing Mediators with signals and slots helps.

4.2.2. Observer

The intent of the **Observer** pattern is to allow several objects to be notified when one object changes state. **Subjects** have several **Observers**. Observers can attach to and detach from Subjects at any time. The Subject maintains the list of Observers, and calls a specific Update() function whenever it's state changes. The Observer can then query the Subject's state by calling GetState().

What Connect helps implementing

Connect implements the Observer pattern *directly*. Attaching, detaching, updating can all be replaced with signals and slot. In fact, Connect can do more: it can completely

decouple the state's source from the it's Observers. Moreover, we can connect objects in different domains by using adapters.

How Connect can be used with this pattern

Define signals for the Subject, emit them when the internal state changes. The parameters for these signals should carry enough information for the Observers.

On the other end, define methods in Observers which can be suitable for slots. Connect the signals to the Observers' slots.

5. GENERSYS – An Example Of Use

In this chapter, I will give an example of how Connect could be used to help implement a real world application.

Applications *can* do without the features of Connect, and the one that I will use for demonstration works perfectly, if with a few quirks. I will conclusively show how Connect can be applied to this application with minimal overhead, to work out some inconsistencies, or make implementing new features simpler.

In the following chapter I will describe some of the functions of the software, and make an issues list, marked with roman numbers in parentheses (e.g. *(i)*). Then in the next chapter I will demonstrate how these issues can be solved with Connect.

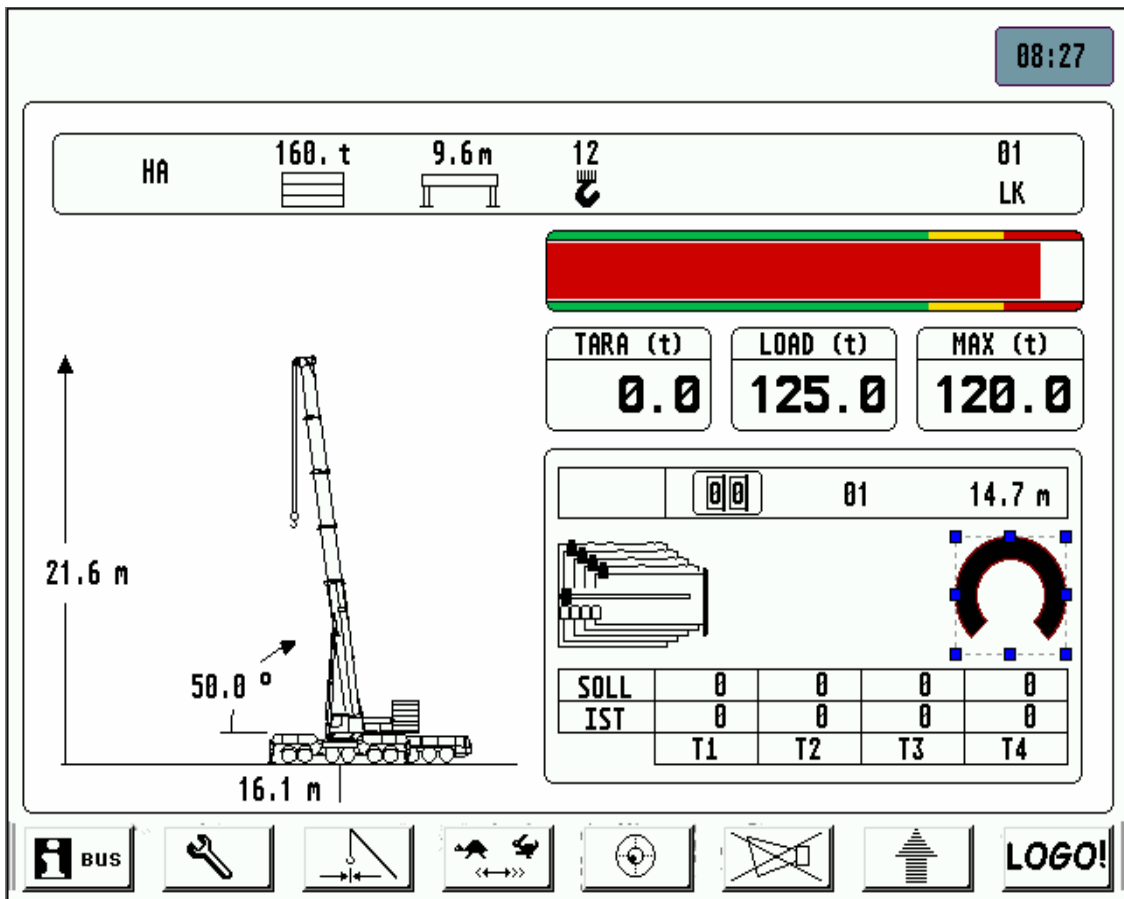
5.1. Description of the Software

The application, Genersys, is a CAD software, used to design custom electronic controllers. Physically these controllers have an LCD screen, some function keys, and a numeric panel. The controller connects to the machine with a bus.

The LCD screen is either monochrome or color, with a resolution of 640*400, 600*400 or 640*480.

The function keys are arranged in rows and columns. Basic controllers have 8 function keys in one row, advanced controllers could have more rows.

The bus sends commands and signals to the machine (e.g. to operate motors) and receive information from sensors. All this data goes into a Process Database – the set of variables used by the controller.



The operator of the machine is presented with screens similar to this. The pictures at the bottom of the screen are the function keys. These function keys can invoke many actions: they can be used to navigate between screens, to start or stop a motor, or to exchange the current set of keys for another predefined set.

The screens may contain several items: icons, bars, real and integer numbers, etc. These items display variables from the Process Database.

GENERSYS, the software, lets the user design the data used by the controller: it lets you define Variables, Function keys, Key sets (a predefined set of keys which can be loaded at once), Symbols, Fonts, Filling Patterns, Subscreens and Screens. Following is a description of relevant attributes of these objects.

Variables are identified by their *name* (up to 6 characters), they have a *type* (bit, byte, real or word), and several other attributes (such as limits and bus addresses) which are unimportant for Connect.

Function keys have a *name* (up to 6 characters), an *icon* and an *action*. The action has a parameter, either a *Screen name*, a *Key set's name*, or an *integer code*.

Key sets have a *name*, and a number of *keys* (references to Function keys).

Symbols have *name*, and one or many *icons*.

Fonts are used to display text and variables. They have a *name*, an *index number*, a *size* and a *bitmap*.

Text modules are lists of texts. There's one module for every language.

Filling patterns have an *index number* and a *bitmap*.

Subscreens and **Screens** have a *name*, a *number*, a number of *keys* (references to Function keys), either a *background bitmap* or a *reference to another screen* which provides the background for this screen, and a number of *widgets*.

Widgets display variables variables in a number of ways, using different objects as tools. Every widget belongs to a single *screen*, has a *variable*, a *position* and a *size*. **Numeric** widgets display real or integer variables numerically, with a *font*. **Icon** widgets display their variable graphically with a *symbol*. **Bar** widgets display a horizontally or vertically growing bar using the *fill patterns*. **Text** widgets display a line of text from a Textmodule.

GENERSYS allows the user to create, delete and modify Variables, Keys, etc. and place them on the Screens.

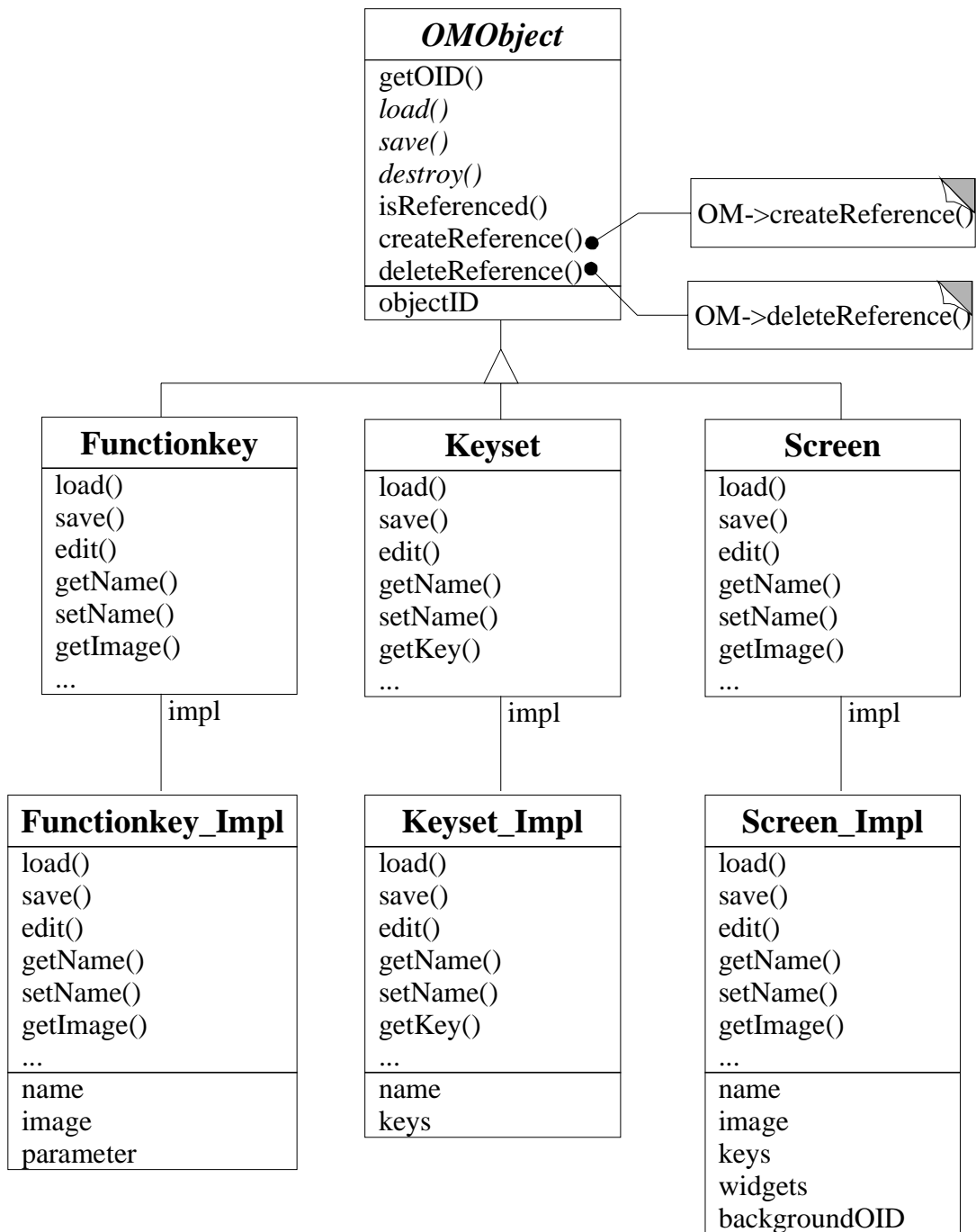
5.2. Original Architecture

The design of Genersys originates from 1995. It is a C++ application, written in Borland C 4.0. I will now outline the design of Genersys. I have simplified the design to a degree, leaving out many parts which are unimportant for the discussion. Also I have renamed the classes, methods and attributes, because Genersys has a mixed English and German source code.

Genersys has an **object manager** (OM), a central object responsible for managing the loading, unloading, saving of every other object. The OM also keeps track of **references** between objects. References in Genersys mean that the referrer needs the referred object for something, and therefore the referred object mustn't be deleted. Every object is identified by its **object ID** (OID), which is in effect a proxy to implement a smart pointer to the object.

The OM also has a kind of version control system.

All of the objects in Genersys are descended from the abstract **OMObject**. Although Subclasses of OMObject use are thin interface classes and use an implementation class to hide the internals, I left that out from the diagram to keep it simple. Attributes such as **keys** or **widgets** store a list of OID's.



Communication problems in the domain

One of the key problems is that there is no facility which enables a referred object to notify its referrers (observer pattern).

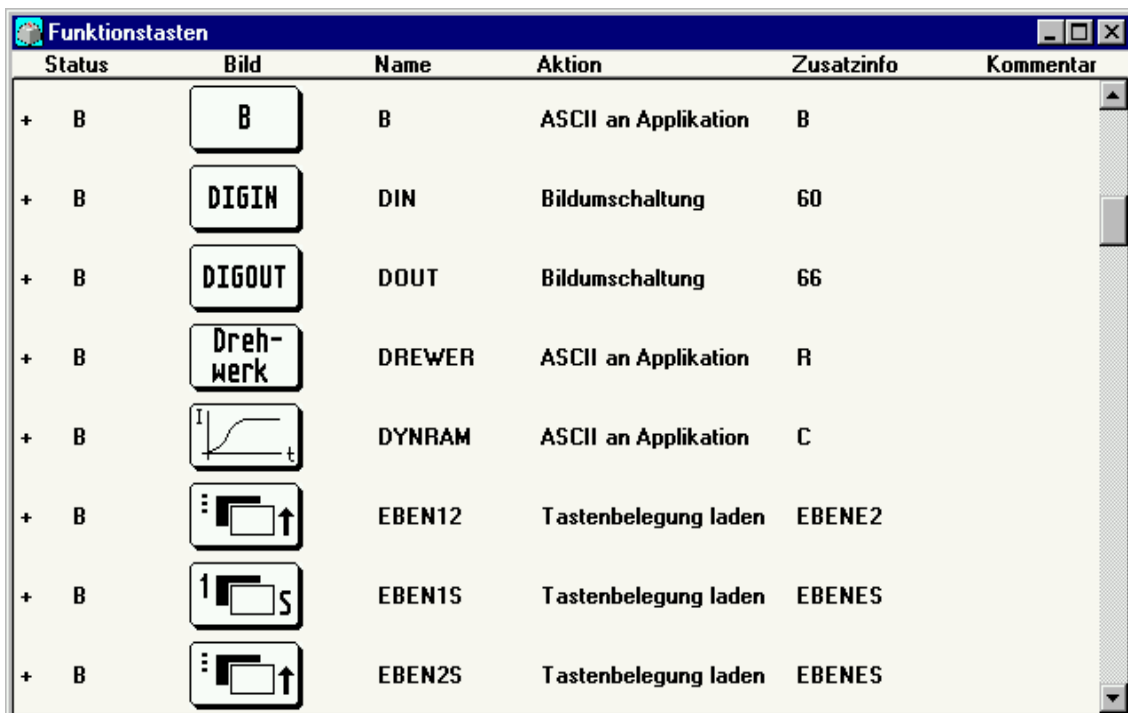
(i) For example, if a Functionkey's icon changes, the graphics in the referring Screens' and Kersets' windows remains the same until it is repainted (for example, dragged off-screen and back again).






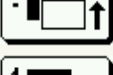


(ii) Worse is the case of Symbols. Unlike the functionkeys, which have a fixed size, the Symbols can change their size. The Widget symbol won't be notified of changes of the Symbol that it uses. Consequently the Symbol widget will have an incorrect size.

(iii) Fonts too can change their size and face. Fonts are used directly by Textmodules and Numeric Widgets, and indirectly by Text Widgets. For this reason, a *referenced* font has its editing completely disabled.

(iv) Same problem arises when a Screen's background is used by another Screen. The Screen's background cannot be modified if the Screen is referenced.

Crosscut problem: object listing



Status	Bild	Name	Aktion	Zusatzinfo	Kommentar
+ B		B	ASCII an Applikation	B	
+ B		DIN	Bildumschaltung	60	
+ B		DOUT	Bildumschaltung	66	
+ B		DREWER	ASCII an Applikation	R	
+ B		DYNRAM	ASCII an Applikation	C	
+ B		EBEN12	Tastenbelegung laden	EBENE2	
+ B		EBEN1S	Tastenbelegung laden	EBENES	
+ B		EBEN2S	Tastenbelegung laden	EBENES	

Genersys provides a list for every kind of object. Objects are added to the list when they are created, and removed when they are deleted. The list shows some of the object's attributes. The list, however, is not notified when the object changes. Instead, the list is made the *single access point* to the editing functions of these objects. It is by the menu of this list alone, that objects can be created, modified or deleted, and the list updates itself after every one of these operations.

(v) The list of Key sets has a bug: if the user modifies a key in the Functionkeys list, the Key sets which use it are not updated in the Key sets' list.

(vi) It would be nice to offer the user the ability to modify or create symbols directly from the Symbol Widget's editing dialog, but it is not possible due to this design.

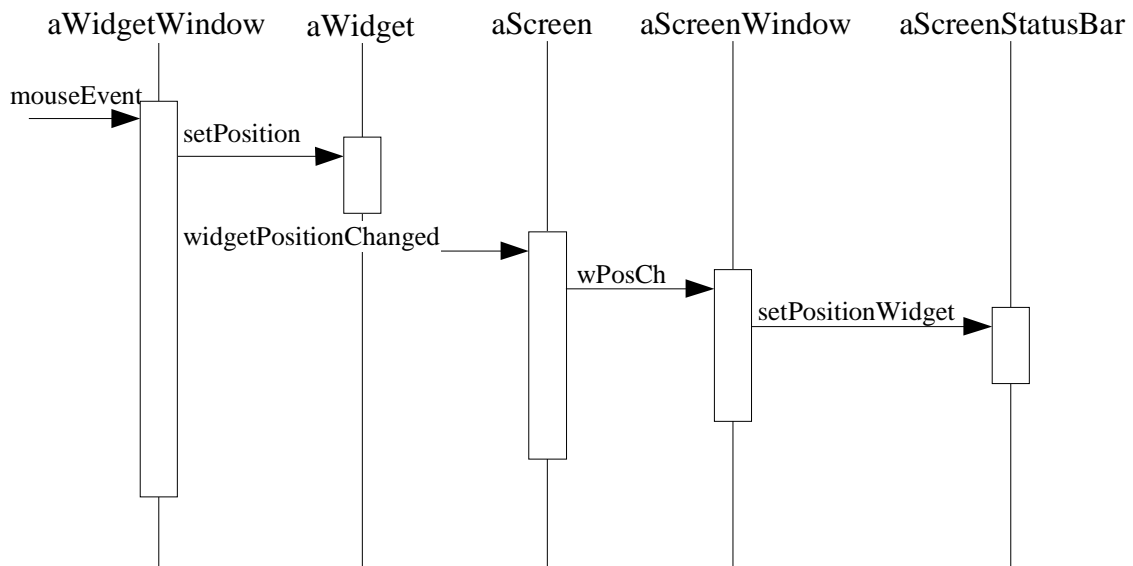
(vii) Another consequence is that objects can belong to *one list only*, although it might be useful to provide additional views, such as a list of objects, like a list of object that were modified since the last save, or a list of object that were created in the current version.

Crosscut problem: user interface elements

The Screen editor window, `ScreenWindow`, is one of the most complex parts of the Genersys user interface. It has menu items to load or choose a background image, add various Widgets to the Screen, move these widget around with the mouse or keyboard, edit the Widgets by double-clicking them, etc.

Each Widget has its own `WidgetWindow`. The Widgets have knowledge of their `WidgetWindow` and vice versa. Likewise, the Screen and its `ScreenWindow` reference each other. The `ScreenWindow` has access to some other GUI components, such as the menu bar and status bar of the application window. The status bar has bits that display the widget's position, size and some other miscellaneous information.

Due to the lack of a mediator and signals, sometimes the communication diagram of these participants gets very convoluted. Observe what happens when the user drags a Widget to a different position:



For a drag-and-resize operation the same would happen twice.

`widgetPositionChanged` and its fellow functions need to be called manually at every operation where the Widget's position changes. Mouse events, keyboard events, the editor of the widget all initiate the same call.

(viii) With all this communication hardwired by tightly coupled objects, it is difficult to implement operations such as moving multiple Widgets at once, automatically align Widgets, or snapping Widgets to a grid.

5.3. Redesign with Signals

What Genersys needs is applying the Observer and Mediator patterns.

For example, the lists need to observe the objects which are listed in them, in order to automatically update themselves. All of the objects need to be made observable. Some

other graphical elements, such as the widgets that display the Functionkeys need to observe the Functionkeys. The list goes on.

First I will describe the steps required to add Connect to GENERSYS.

1. First, we add signals to every implementation class with Connect. For every attribute X we add an XChanged() signal. For example, the Functionkey_Impl class shall have an imageChanged() signal.
2. We insert calls to the XChanged() signals at appropriate locations. For example, the Functionkey_Impl::setName(string) function compares the new name to the old name, and if they are different, it emits the nameChanged() signal.
3. Every descendant of OMOBJECT is made the signal proxy of its implementation, this code goes into the implementation constructor.

Now let's see how Connect helps with our issues list.

- (i) **Functionkey widget icon updating:** Connect the iconChanged() signal from the Functionkey to the widget's setIcon() slot.
- (ii) **Symbol size and icon updating:** The Widget symbol should have a new method, symbolChanged(), to deal with changes to the symbol. Connect this to the symbol's signal.
- (iii) **Font updating:** Fonts' signals need to be connected to every numeric Widget and every Textmodule. The Textmodules in turn need to be connected to the text Widgets.
- (iv) **The screen background update:** Connect every Screen editor widget's updateBackground() slot to either it's own Screen's signal, or in case the Screen uses another's background, that other Screen's signal.
- (v) **Key set list updating:** Connect the Functionkeys' signals to this list too.
- (vi--i) **Modify anywhere:** Lists no longer need to own their elements. The element's signals will notify when and how they need to update themselves.
- (vi--ii) **Multiple listing:** See above. An element can be connected to multiple lists.
- (vi--iii) **Screen widgets:** Simply connect the positionChanged() signal of the widget to the statusbar widget. Probably requires an adapter.

This is as simple as it sounds.

6. Conclusion

Some concluding thoughts on Connect.

6.1. Comparison to Others

First, let's see how connect compares to other signal/slot systems, in the C++ area.

	<i>Connect</i>	<i>Qt</i>	<i>LibsigC++</i>
Properties			
Requires special compiler steps	Yes	Yes	No
Signals are	Methods	Methods	Objects
Slots are	Objects	Methods	Objects
Connections stored in	Connect	Object	Signal
Requires templates	Yes	No	Yes
Features			
Automatic type conversion	No	Yes	No
Any function can be slot	Yes	No	Yes
Void returns	Yes	Yes	Yes
Non-void returns	No	No	*
Adapters	Yes	No	Yes
Transitions	Yes	No	Yes
Signal proxies	Yes	No	No
Compile-time type checking	Yes	No	Yes
Every signal costs memory	No	No	Yes

Qt and Connect require special compiler steps. Qt has a *metaobject compiler* which does several tricks, including inserting signal code. Connect's weaver also requires special steps. LibsigC++'s signals are objects, not methods, and can therefore be in the global scope. This, of course, means that every signal of every object costs memory, used or not.

Qt and LibsigC++ store connections decentralized. Connect stores connections in one object, which in turn allows for the creation of signal proxies. As we have seen, signal proxies are crucial in implementing several design patterns. I would say that these proxies are the greatest achievement of this whole work. Neither Qt, nor LibsigC++ or anything else that I have seen in my studies has constructs as powerful as signal proxies.

Finally, the question of non-void returns. Neither Qt nor Connect allows slots to return values. (*) In fact, LibsigC++ only guarantees a correct returned value if the signal is

connected to exactly one slot, and in the case of zero or many connections the returned value is undefined. The original idea of Connect (notifying objects in other domains of the change of then internal state of an object) doesn't need returned values. Also, sending out commands (the other use of signals) doesn't necessarily require acknowledgment. In either case, Connect can be hacked into supporting returned values, e.g. by passing references as parameters.

6.2. Future Improvements

I have many ideas of how Connect can be extended and improved.

Public signals

For the time being, all the signals are *protected*. It may be useful to be able to declare certain signals as public. The **SIGNAL** tag could be extended with a **SCOPE** attribute, which declares the signal public or protected. This is very trivial to implement, I left it out because I wanted this version of Connect to be as clean and small as possible.

Signal blocking

Some or all signals from a given objects could be blocked and unblocked at runtime. Qt has this functionality, and it's quite useful to avoid signals being sent while a group of objects are in a transient state in the middle of an update operation.

Asynchronous signal delivery

The current implementation of Connect delivers signals immediately, as soon as they are emitted. This may not be desired, especially if the delivery of the signal causes other signals to be emitted. The other signal can precede the delivery of the current signal to some slots. Since the order of delivery is undefined, this can lead to unforeseen interference (race conditions, etc).

Asynchronous delivery of a signal would mean that the signal is emitted, but the calling of the receiving slots is delayed until the system has processed some pending events. This can be very desirable. Imagine that the backend (the domain of the application) is doing some complicated update process. We wouldn't want the GUI to always update to the transient states during the update. Asynchronous delivery could even merge some of the signals before delivery.

Any signal could be declared asynchronous in the Connect file, or maybe the connection itself could be created as asynchronous at runtime.

Network transparency

As noted in chapter 1.4.3, distributed computing is a common crosscut problem. Slots are already doing a fine job of an abstract callback. This concept could be extended to handle signals between remote computers. With a few additional factories added, Slots that handle Remote Method Invocation can be added to Connect.

I have seen aspect-oriented solutions that weave network transparency into application at the class level. Network transparency with slots could provide a lighter solution, and readily used with existing code.

Integration with frameworks and IDEs

Aspect-oriented programming has potential, but potential is useless without powerful tools. The greatest strength of OOP lies in the huge array of reusable components (frameworks, OOP libraries, widget sets, etc) that are available off the shelf.

Connect could be the small, invisible basis of an aspect-oriented application framework that could handle various crosscut problems.

7. Acknowledgments

There were a few people who have helped me immeasurably, and without whom this work would never have come to be. I would like to thank my parents and grandparents for their help and support, I am truly grateful. I also need to thank all my friends and colleagues to whom I have explained (or tried to explain) the idea of Connect countless times. Every time I told someone new about Connect helped *me* better understand it. Also the hints and ideas that I have received helped Connect mature. Special thanks to Tóth Mariann and Pápai Miklós for all the criticism and inspiration. Also thanks to all the open source hackers who have written all the software tools that I have used writing this. Last, but not least, thanks to my consultant, Frigó József.

Appendix A. Bibliography

JAR95: , The Jargon File, 1995,

DAHL72: O.-J. Dahl, W. E. Dijkstra, C. A. R. Hoare, Structured Programming, 1972

KON97: Dr. Kondorosi László, Dr. László Zoltán, Dr. Szirmay-Kalos László,
Objektum-orientált szoftverfejlesztés, 1997

GAM95: Gamma, Helm, Johnson, Vlissides, Design Patterns, 1995

Ciu01: Oliver Ciupke, Objektorientiertes Design unter der Lupe, 2001 FZI

Kic97: Gregor Kiczales, Aspect-Oriented Programming, 1997

Mon00: Richard Monson-Haefel, Enterprise JavaBeans, 2nd Ed., 2000

Nie97: Niemeyer, Peck, Exploring Java, Second Edition, 1997

REC01: Andreas Ludwig, RECORDER Technical manual, 2001,

COM01: Dr. Uwe Assmann, COMPOST White Paper, 2001,

ASPJ98: Tracy Kugelman, AspectJ Project Fact Sheet, 1998

XER01: Hilsdale, Kiczales, Aspect-Oriented Programming with AspectJ, 2001

Kut99: V. Kuttruff, Inject/J - Ein Werkzeug zur skriptgesteuerten
Quelltexttransformation, 1999

Appendix B. Floppy Contents

The software that were described in this thesis is available on the floppy insert. It is also available at my website, at

<http://apocalypse.rulez.org/~upi/Connect>

The code is covered by the GNU General Public Licence, available at

<http://www.gnu.org/licences/gpl.html>

Subdirectories on the floppy:

- **\CPP:** C++ code parts and the weaver.
- **\CPP\include:** C++ include files.
- **\CPPEXample:** C++ demonstration project.
- **\CPPEXample\before:** The source code before weaving.
- **\CPPEXample\after:** The source code after weaving.
- **\Java:** Java code parts and the weaver.