Decoupling of Object-Oriented Systems

A Collection of Patterns

Version 1.1

Jens Coldewey

Coldewey Consulting 17.07.00 (revision)

(based on material written for sd&m AG, Munich in 1996)

Please observe: This paper contains work from 1996, I would consider *partly outdated*. The document cries for several changes, such as renaming of some patterns and some more negative consequences, but this is not my focus currently. I hope the document is still usable.

Jens Coldewey, July 2000

Coldewey Consulting
Uhdestr. 12
D-81477 München
Telefon +49-(700) COLDEWEY (+49-700-26533939)
Telefax +49-89-74995702
email: jens_coldewey@acm.org
http://www.coldewey.com

Decoupling

When you analyze object-oriented projects in trouble, you often find designs like the one depicted in Figure 1. The team designed the logical class model correctly but started implementation without a physical class design. The result is a strongly interwoven net of dependencies, that may cause severe problems during compilation and testing: Because nearly every class depends on every other, any minor change will start a recompile of the complete system. Separate testing of classes is impossible. It is hard to reuse classes from this design in other projects, because you will not be able to cut them out of the dependency network, without rewriting most of the code. The design shows a phenomenon, Bruce Webster calls *Hyperspaghetti Objects and Subsystems* [Web95].

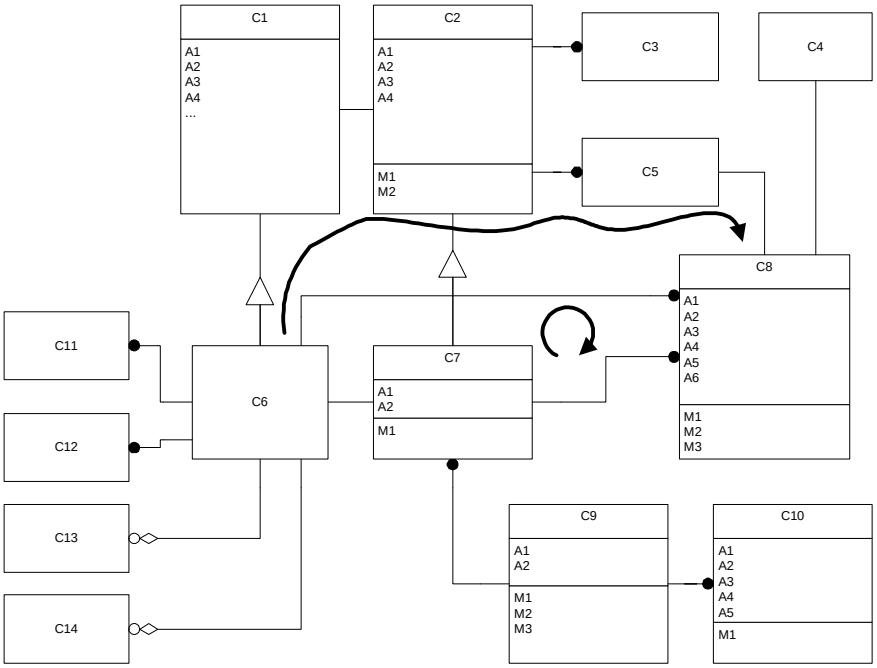


Figure 1: A pathological example of an object relation graph, taken from a real-world project. Several redundant and cyclic relations between C6 and C6 cause tight transitive coupling of all classes.

Object-orientation encourages you to break down the code into very small methods, Smalltalk programs have an average method length of 1.5 lines. The gain is high reusability but you pay with increased interaction between methods. The complexity shifts from the single method's control flow to the interaction complexity of the object model. These interactions couple classes tightly and tight coupling has several negative consequences:

- Particularly with strongly typed languages, such as C++, the compiler parses every class in the web to ensure type correctness. At best, this behavior may result in make-dependencies that cause a recompilation of the complete system after each minor bug fix. At worst, the compiler or linker runs against some internal limitation and refuses to build the program.
- It is difficult to isolate a group of classes for component tests. Tracking down errors will be a hard and time-consuming job.

- The program is hard to maintain. Changes propagate all through the system and it is difficult to determine the responsibilities of a class. Therefore classes blow up with dozens of attributes and methods.

To prevent this mess you have to manage the dependencies between the classes. This paper contains a number of design patterns that help you to do so.

Some Definitions

Subsystems

To fight relation complexity you have to identify *subsystems*[1]. According to Grady Booch, a subsystem is "a physical structure that can be released and for which engineers can assume responsibility". In other words, a subsystem is a chunk of cohesive classes that can be compiled, linked, and tested separately[2]. Note, that a subsystem is not a single class but contains several classes. Some languages offer support for subsystems, others require to take special measures to ensure separation.[3]

Robert C. Martin gives three rules of thumb that may help to identify subsystems:

> To be cohesive, the classes in a [subsystem]
>
> 1. Must be closed together
> 2. Must be reused together
> 3. Should share a common function or policy
>
> The order of this list is significant; the first rule is more important than the second, and the second is more important than the last. [Mar95, p. 241]

Classes are *closed together*, if a change to the application effects either all or none of them. Note that you need an idea of what changes are likely in the future to determine whether a subsystem is closed or not.

Reusability is a good check whether a subsystem is well-chosen. "Having common reusability means that the classes are inseparable. Any attempt to reuse any class within the [subsystem] will cause all of them to be reused" [Mar95, p. 241].

Static and Dynamic Coupling

To analyze dependencies we have to distinguish between *static* and *dynamic* coupling. Defining static coupling is easy, when you are working with statically typed languages: Two classes are *statically coupled* if you need the definition of one class to compile the other.

---

1 Subsystems are also known as *modules*. However, a module also is an implementation concept for subsystems in 3GLs, so we use the more general term of subsystems.

2 Booch also defines 'Categories' which are "clusters of classes that are themselves cohesive, but are loosely coupled relative to other clusters." Since Subsystems and Clusters correspond in most cases, we follow the suggestion of Robert C. Martin, to merge both. Still we use the more graphical term of a 'Subsystem' instead of 'Category'.

3 Unfortunately, popular languages, such as Smalltalk and C++, belong to the second category, while the first group contains rarely used languages such as Modula-3 and Oberon.

With dynamically typed languages such as Smalltalk we define static coupling as the knowledge you need during programming to use a class or a subsystem.

Two classes are *dynamically coupled*, if their instances send messages to each other at run time.

The distinction between static and dynamic stems from polymorphism. When you write code that calls a method of a object, you never know whether you call the method of the reference's class or rather a method of its heirs. Hence a static relation to a single class may cause dynamic relations to objects of all its heirs.

Static coupling is the more interesting design parameter of these two, because we are interested in chunks of code we can develop independently. Controlling and reducing static coupling is a good strategy to avoid a "Hyperspaghetti" system. However, static coupling offers the compiler the possibility to check for correct interfaces. Therefore, reducing static coupling may reduce type safety. Dynamic coupling joins into the game as soon as you start testing. To provide good testability you also should reduce dynamic coupling.

Transitivity

Transitivity is an issue that comes with static coupling. The interface of a method also contains the parameters you need to supply. In most cases these parameters are not atomic datatypes but complex classes, which you also have to know. Thus the interface of the method transitively contains other classes which in turn may contain further classes. You often find circles in the transitive closure of a class which means, that every class participating in the circle needs every other class to compile. This is one of the most unpleasant situations you can encounter. Some languages, such as C++, add fuel to the difficulties because they do not separate interface and implementation of classes cleanly. C++ class templates are the worst example because you need the complete implementation to instantiate them.

Coupling and Collaboration

There are several ways, subsystems may interact and these different ways influence coupling differently. Since subsystems are built from classes, they communicate like classes. They may send a message to another subsystem[4], inherit from it or use common data. Apparently, any of the preceding alternatives results in a different quality of coupling:

- *Using data of another subsystem* needs knowledge about its interior design. If you change the subsystem that provides the data, you also have to take care of the other subsystems. Since you can also use memory pointers to access data, a later dependency analysis may easily become impossible. On the other hand, there is nothing you can do with shared data you cannot do with message passing, so avoid shared data

[4] Sending a message is a more abstract concept than just calling a method. The latter is only one possibility to implement message passing. Other option include message queues of the operating system, signals, or object brokers, to mention just a few.

- *Message passing* needs the least information, you only have to know the contract of a method: its name, its parameters, the calling mechanism, and the pre- and post-conditions [Mey88]. Information about the subsystem's implementation is not necessary, so you may arbitrarily change the interior of the called subsystem without any effect on the caller. Still, to pass a message to an object you have to reference it.
- *Inheritance* is somewhere in between. The degree of coupling depends on the way you use inheritance [Gam92, pp. 6-7]: *Specializing* a class of another subsystem needs deep knowledge and therefore results in tight coupling. If you inherit only an *abstract protocol*[5] you do not need to know any details of the other subsystem. Some of the patterns of this chapter use abstract protocols to decouple subsystems. Note, that inheritance raises different coupling qualities, depending on its usage.

A Strategy for Decoupling

From this discussion we can deduce a strategy for preventing hyperspaghetti systems:

1. Identify subsystems
2. Reduce static coupling between different subsystems' classes. Pay special attention to transitive coupling.
3. Reduce dynamic coupling.

Suitable combinations of collaborations reduce coupling but often introduce additional classes or have other drawbacks. There are several design patterns that describe proven combinations. Still, they only help if the subsystem division reasonably reflects the problem structure of your system.

[5] *Abstract protocols* are classes that contain only abstract methods. These methods comprise the interface, a client has to provide if it uses the subsystem. Hence, a client class has to inherit and implement the abstract protocol. This is a good way to guarantee complete interfaces.

Pattern Overview

This chapter contains five patterns that deal with different aspects of decoupling (Figure 2). Every pattern is either an application or a combination of design patterns from [GOF95].

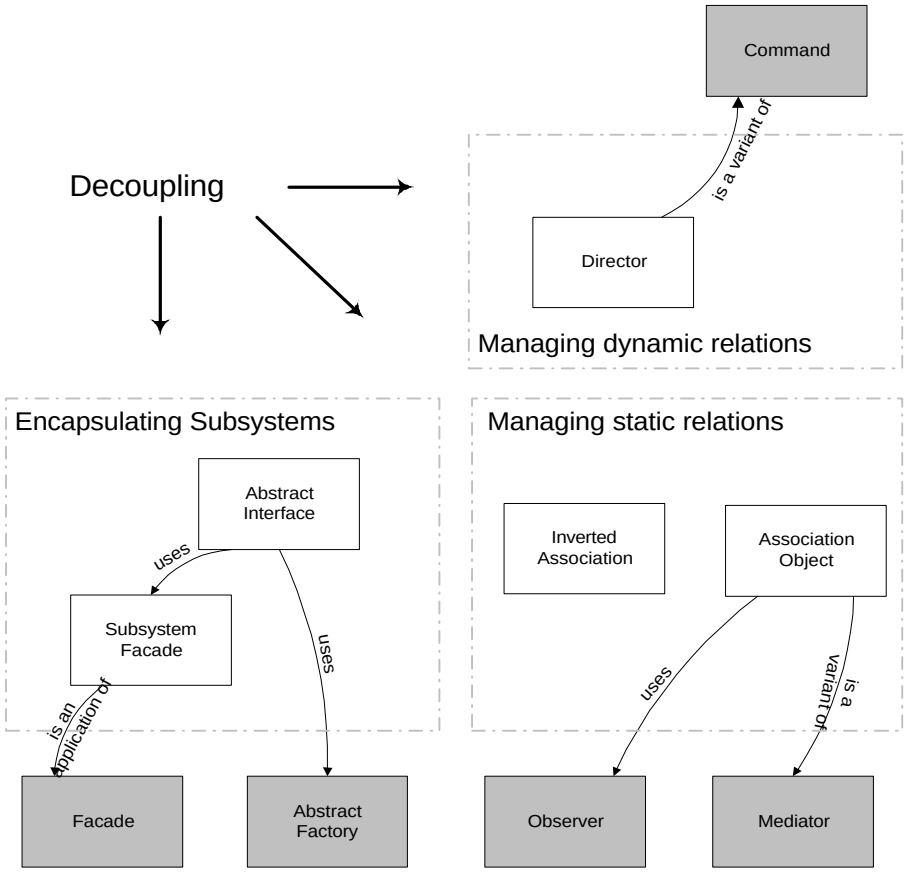


Figure 2: A map of the pattern language. White boxes indicate patterns of this language while gray boxes indicate patterns of [GOF95].

- Patterns to *encapsulate subsystems* give hints on how to shield complex subsystems. They allow you to have arbitrarily complex (or simple) sets of classes in a certain subsystem.
- *Management of static relations* provides ways to break cyclic dependencies and to decouple objects, statically coupled.
- To cope with *dynamic relations* the Director pattern shows how to model complex use cases, while preserving a loose static and dynamic coupling of the objects.

Further Reading

Robert C. Martin gives an excellent discussion of how to compose subsystems and what to consider during this process [Mar95]. The book also contains an ‘experimental metric’ to measure, whether the subsystems of a given system are well balanced or not. His series about principles of OO-Design [Mar95b], [Mar96], [Mar96b], [Mar96c] and [Mar96a] also focuses on subsystem identification and decoupling.

[Gam92] demonstrates the usage of design patterns for decoupling of subsystems in the ET++ framework. It is a good example of how to combine different patterns to get good decoupling

Jiri Soukup presents an interesting discussion about the similarity between object relations and jumps in procedural languages [Sou94]. He suggests to structure relations between objects, much as structuring jumps helped dealing with spaghetti code. As solution he presents ‘pattern classes’ (not to confuse with our notion of patterns[6]). According to Soukup every cyclic dependency should be eliminated with this classes.

Martin Hitz and Behzad Montazeri provide a set of measures to analyze coupling and coherence [HMo96]. The article contains a good survey about metrics for coupling. While the work is preliminary and does not address more complex problems, it is still a valuable resource of forces effecting coupling and coherence.

6 In fact, Soukup included several pages of discussion about the relation between design patterns and his pattern classes.

Subsystem Facade

Also Known As

Facade [GOF95]

Abstract

Subsystem Facade is a special variant of the Facade pattern. It encapsulates a complete subsystem to ensure separate development without any restrictions on the internal design of the subsystem.

Example

Consider the Instrument Interface of a Medical Laboratory Manager. Adapting the technical interface of a certain instrument to the manager is a complex task. There are high level jobs, such as interpreting messages coming from the instrument or formatting commands of the Lab Manager. Other low-level tasks concern buffering of data, error logging or administration of configuration data.

Figure 3 shows several classes of this subsystem, which may vary from instrument to instrument. Client classes, addressing a certain instrument, have to cope with the complete web of objects.

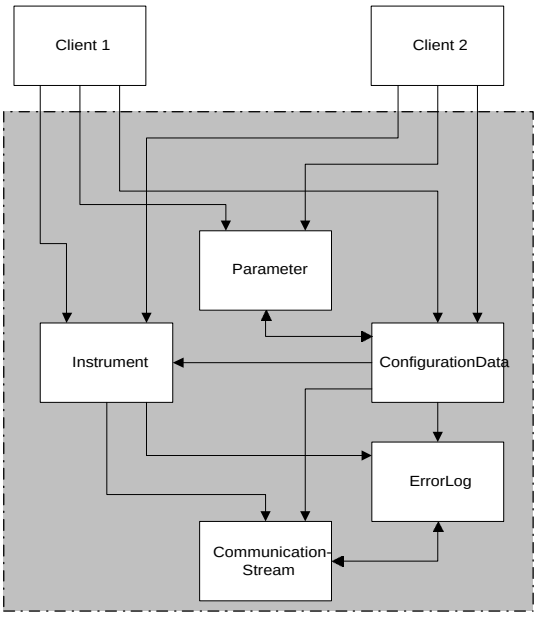


Figure 3: The instrument interface without decoupling measures

One solution to this problem would be a redesign of the instrument interface to implement a strictly layered structure. This is a reasonable way, if the instrument interface is a large subsystem with dozens of classes. However, with smaller subsystems the effort may not pay off.

Context

A medium complex subsystem needs minimization of coupling to other subsystems. However, you do not want to redesign the subsystem, because the effort would not pay off.

Problem

How do you encapsulate an arbitrary subsystem without changing the interior?

Forces

- *Decoupling versus effort:* If classes are well decoupled from one another, they are easier to maintain and easier to reuse. However, the domain structure of the subsystem often results in tight coupling. In these cases decoupling usually means to add extra classes or to redesign the subsystem. Both solutions are expensive.
- *Interface complexity:* A complex subsystem interface is hard to use. Interior changes of the subsystem propagate to the exterior clients and enforce unexpected changes. Additionally, complex interfaces result in complex and less efficient tests.
- *Connecting to legacy systems versus simple interfaces:* If the subsystem is legacy software, you may not be able to change the code. On the other hand, these systems often present complex interfaces, which provide much more functionality than you need.

Solution

Do not touch the subsystem. Instead, find a good abstraction of the Subsystem. Build a Subsystem Facade class that interfaces the subsystem to the clients according to the abstraction. Hide tight coupling inside the subsystem.

Structure

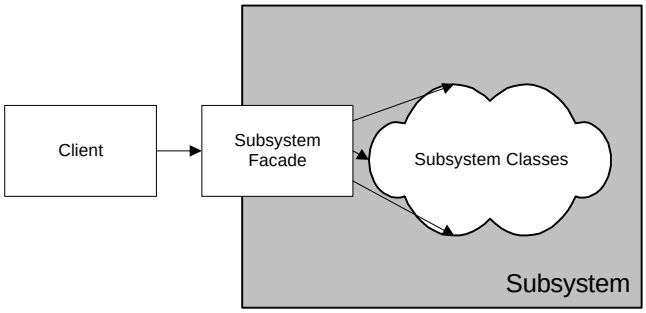


Figure 4: The structure of a Subsystem Facade. A single class completely encapsulates an arbitrarily structured subsystem. Clients have access to only this class.

Participants

- **Subsystem Facade**
  - encapsulates the class net of the subsystem
  - delegates clients requests to the appropriate objects of the subsystem
- **Subsystem Classes**
  - process the requests
  - have no notion of the Subsystem Facade

- **Client**
  - only knows the Subsystem Facade
  - directs all requests to the Facade and gets all results from there

Dynamic Behavior

Clients direct their requests to the Facade, which forwards them to the appropriate subsystem classes. The Facade also manages the return of results.

Example Resolved

Figure 5 shows the instrument interface, decoupled with a Subsystem Facade, called *InstrumentInterface*. The two clients access only one class instead of several classes in Figure 3.

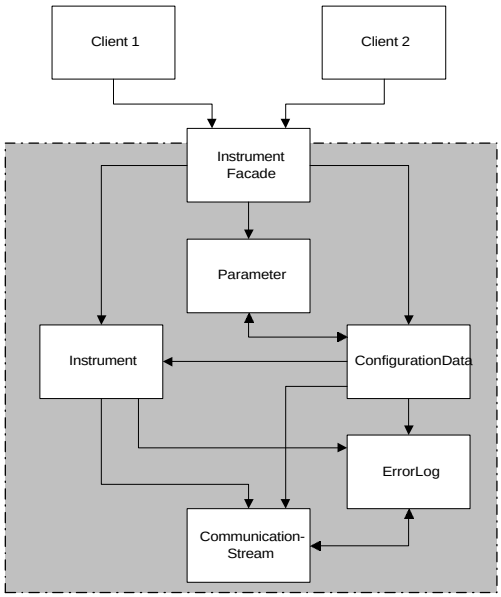


Figure 5: The instrument interface subsystem decoupled with a Facade

Consequences

- *Decoupling:* The Subsystem Facade reduces external coupling. Since all external classes reference only the Subsystem Facade, they are statically and dynamically coupled to only one class. A well-implemented Subsystem Facade also cuts the transitive closure (see Implementation). This eases separate development of the subsystem as well as testing and maintenance.
- *Effort:* The coupling between Subsystem Classes may be arbitrarily tight. The Subsystem Facade encapsulates the complete net inside the subsystem. Hence you may ignore coupling issues inside the subsystem. With small subsystems this may save considerable effort. However, larger subsystems may still develop coupling problems. Therefore, this pattern is only suitable for small subsystems unless you take additional measures inside.

- *Connecting legacy systems:* Since the Subsystem Facade completely hides the interior of the subsystem, it may as well hide legacy code. The Facade may present an interface that conforms to a new system design, while the subsystem is still old code. This approach is useful to do a step-by-step migration.

Implementation

- *Instances of subsystems:* You may interpret the Subsystem Facade as an abstraction of the subsystem, modeled as a class. Depending on the domain aspects you have two alternatives to choose from: The Facade may either be a Singleton or you may decide to allow multiple instances of it. Which one is correct depends on the abstraction the subsystem models. If the subsystem implements communication issues, for example, every instance of the Facade may represent a separate connection. If the subsystem is a book-keeping system and the Facade only offers use-cases, it may as well be a Singleton.
- *Instantiation of subsystem classes*. There are three alternatives to instantiate objects inside the subsystem:

    1. *Clients are responsible:* To preserve encapsulation, the Facade needs special methods to create the corresponding objects. This solution slides towards the Abstract Interface.

    2. *Facade is responsible:* When the Facade is a Singleton [GOF95] you may make it responsible to create new objects inside the subsystem. The clients are unaware about creation and deletion of subsystem objects. This solution works best, when multiple clients address a single server with a stateless protocol. Be sure that the Facade is the only way to access the subsystem.

    3. *Subsystem objects are responsible:* There are factories inside the subsystem that know, when to instantiate certain objects and when to destroy them. The Facade only triggers use cases. This is the most radical approach where even the Facade is not aware of object instantiation. This solution works best with autonomous subsystems that also offer some of their services through the Facade.

    Most larger subsystems implement a mixture of these three alternatives. Pure solutions are rare.
- *Subsystem Facade and dynamic link libraries:* If you package your system into dynamic link libraries (DLL), the Subsystem Facade may form an entry point to a DLL. However, not every subsystem is a good candidate for dynamic linking; keep in mind that calling a DLL may cause considerable overhead.
- *Complete versus relaxed encapsulation*: Whether the Subsystem Facade is the only way to access the subsystem or not, is an important decision. Complete encapsulation gives the developer full control over the access to the subsystem and eases testing. However, a subsystem may serve several purposes with different requirements to the interface. It may even be a good idea to have several Facades to a single subsystem, that represent different abstractions.
- *Minimizing compile time dependencies with C++:* If the header file of the Subsystem Facade includes any other header file of the subsystem, the encapsulation is undermined. Use forward declarations to cut the transitive closure. ([Mey92, Item34])

Variants

A larger subsystem shielded with a Facade also is a good candidate for a server in a distributed architecture. Split the Subsystem Facade into a client-side Proxy and a server-side Facade with a Broker between them, such as CORBA or OLE. Minimize the traffic between Clients and Servers. Without a Subsystem Facade you have to define several different Proxies, which makes the subsystem harder to use and to maintain.

Using a Subsystem Facade to reengineer mission critical modules, you may use a smooth migration strategy. The Facade interfaces to the legacy code and forwards requests to new code. It compares both results and generates bug-reports if they differ. This technique allows extensive in-site tests of reengineered code while minimizing the risk and the effort.

Related Patterns

The Facade [GOF95] is a more general variant of this pattern. A Facade may hide an arbitrary piece of software and it does not completely hide the subsystem classes; a client may still access classes without using the Facade.

The Abstract Interface is a special variant of this pattern. It offers additional support, if the client has the control over the life cycle of the subsystem objects. Adapter [GOF95] converts the interface of one class into another interface. Therefore it may also form a Subsystem Facade but this is not its main purpose.

Known Uses

The pattern is an object oriented version of the module concept used with 3GLs [Den91, p. 348]. Most of the 3GL projects at sd&m used this concept.

The DATEV IDVS project, a specialized report generator, used this pattern in two places. It contained a class DsDocument that encapsulated the layout and the preparation of documents. Likewise the DsControl class of the printing subsystem encapsulated all aspects of printing. This class also constituted an interface to another process.

Abstract Interface

Abstract

An Abstract Interface is a special variant of a Subsystem Facade. It encapsulates a complete inheritance hierarchy in a subsystem without exporting subclasses.

Example

Suppose the Laboratory Management Assistant offers configurable tests; the user is allowed to enter new test descriptions, including test procedures, thresholds and a skeleton text for reports Figure 6. A good way to implement the text subsystem is to use a Composite [GOF95] (Figure 7). To enter and edit the texts, there may be a TextController class that implements a text editor. The user may get a scrolled list with possible text elements, such as PatientName.

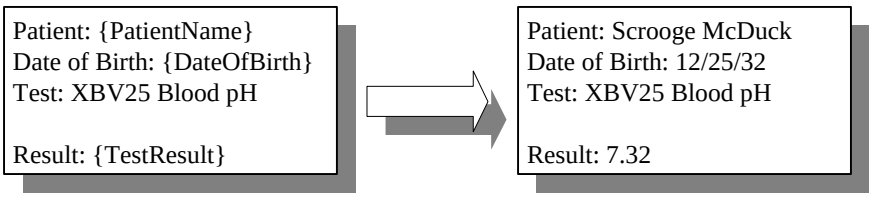


Figure 6: The user may enter a text skeleton, such as the one shown on the left side. The right side shows a skeleton expanded for a report.

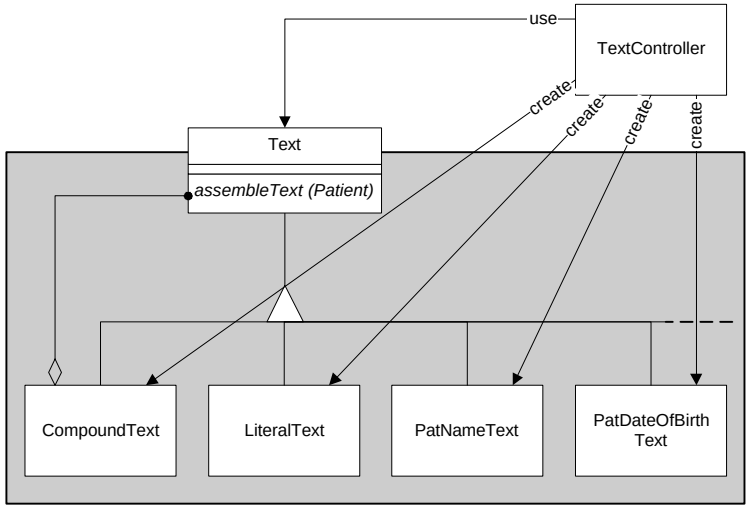


Figure 7: A text skeleton modeled with the Composite pattern.

However, there is a problem creating the text elements, as you can see in the following listing, taken from the TextController:


```
typedef string   textelementDescr;
Text*            t = NULL;
selectionType    s = aComboBox->getSelection();

if ( s == "{PatientDateOfBirth}" )  t = new CompoundText();
if ( s == "{PatientName}")          t = new PatNameText();
if ( matches(s,"\{Date( %[A-z])*\}  t = new FormatedDate;
...
// if none of the previous match, it is literal text
if ( t == NULL )                    t = new LiteralText(s);
```

The TextController has to call the appropriate constructor and thus needs to know all elements available. Now, consider adding a new text element. The TextController needs to know this new class and thus has to change too. It is impossible to ship a set of elements separately without a new TextController. Finally, every additional client of the subsystem has to copy the code. We have violated the *Liskov Substitution Principle*: “Functions that use ... references to base classes must be able to use objects of derived classes without knowing it” [Mar96a]

Context

You want to encapsulate a subsystem, that consists of an inheritance hierarchy. The hierarchy is large or likely to expand. Clients can instantiate different subclasses of the hierarchy. A common protocol is sufficient, to accomplish all other services of the subsystem.

Problem

How do you design the interface?

Forces

- *Decoupling:* To reduce dependencies between the subsystem and its clients, you want to decouple the subsystem as good as possible.
- *Complex subsystems versus complex clients:* A well decoupled subsystem is easy to use and results in simple clients. However, it needs additional methods or classes that deal with object instantiation, making the subsystem itself more complex. Without any encapsulation measures, programming the first version is straight forward and simple but later changes are more expensive.
- *Flexibility versus type safety:* To provide type safety, the client has to know, what classes it works with. As soon as the instance is alive, a supertype is sufficient information to provide type safety. Still, to create a new instance, the client has to exactly specify the heir it wants to get. If it does not know the available classes, there is no way to guarantee type safety during compile time. However, if the client does not know the inheritance hierarchy, you are free to add further classes to the subsystem without changing the clients. This is a particular strong force when the subsystem resides on a central server and the client resides on a large number of remote machines.

Solution

Encapsulate the subsystem using a Subsystem Facade. Make the superclass of the hierarchy the facade. Add a class method to it that creates new heirs depending on a parameter. Use a separate factory class, where subclasses register dynamically. Make the facade delegate creation to this factory.

Structure

Figure 8 shows the structure of an Abstract Interface. The AbstractInterface superclass encapsulates the subsystem. It provides the class method createElement to instantiate heirs. Its parameter determines which ConcreteClass it returns. The Factory performs the creation.

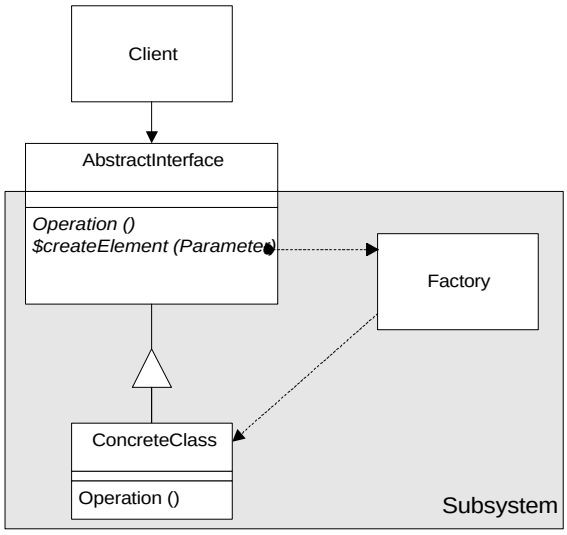


Figure 8: Structure of an Abstract Interface.

Participants

- **AbstractInterface**
  - totally encapsulates the subsystem
  - provides a unified protocol for all classes of the subsystem
  - provides a factory method to create new instances of ConcreteClasses and return the references
  - delegates the creation process to the Factory.
- **ConcreteClass**
  - implements the protocol of the AbstractInterface
  - knows the values of the createElement parameter it is responsible for
- **Factory**
  - knows all ConcreteClasses
  - finds the ConcreteClass responsible for a certain createElement parameter
  - creates the corresponding ConcreteClasses.
  - usually is a Singleton
- **Client**
  - Uses the subsystem only via the protocol of the AbstractInterface.
  - Has no notion of the ConcreteClasses

Dynamic Behavior

Figure 9 shows how a Client instantiates a new ConcreteClass. The AbstractInterface delegates the creation to the Factory which in turn analyzes the parameter and creates the corresponding ConcreteClass object.

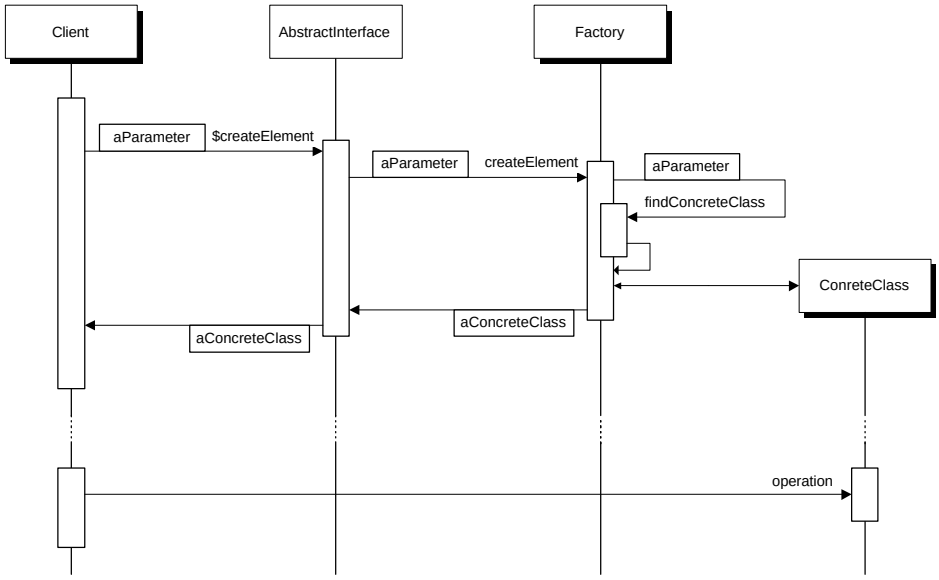


Figure 9: Creating a new ConcreteClass using the Abstract Interface pattern. Instead of directly calling the constructor of a ConcreteClass a client calls the createElement class method of the AbstractInterface. This method delegates creation to the Factory. The shadowless box of the AbstractInterface indicates that this class is abstract.

Example Resolved

Figure 10 shows the text element subsystem designed with an Abstract Interface. The superclass Text becomes the facade of the subsystem. The createTextElement method gets a string as parameter, which identifies the text element. This string may be the Name of a text element, for example " Name: {CurrentDate}". The Text class propagates this parameter to the TextFactory. The factory finds out, that this String requires creation of a ComposedText object with two members: a LiteralText containing ‘Name: ’ and a CurrentDateText.

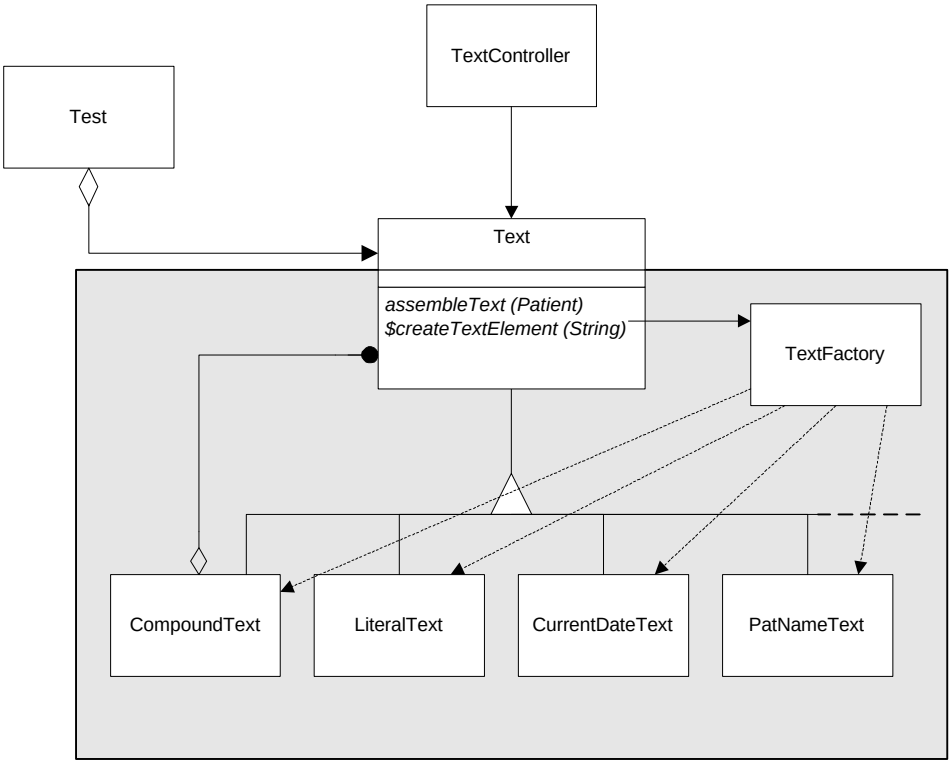


Figure 10: With the createTextElement Factory Method, the Text class completely encapsulates the subsystem. Now a TextParser analyses a parameter and creates the corresponding objects.

The following code from the TextController shows, that the handling has become simpler:

```
Text t;
selectionType s = aComboBox->getSelection();
t = Text::createTextElement(s);
```

Now adding a new text element only needs changes inside of the subsystem. The TextController remains stable.

Consequences

- *Decoupling:* The only interface a client knows is the protocol defined by the AbstractInterface. The Client does not know any ConcreteClass.
- *Complexity:* Since the clients know only one class, they are very simple. However, the subsystem itself is more complex than with a direct approach. Particularly the Factory class is new. It may become complex, when you decide to use a dynamic registry. Still, the Factory is highly reusable, so check your class libraries and frameworks for a suitable implementation.
- *Flexibility:* If the ConcreteClasses register at the Factory you can add new ConcreteClasses dynamically. The client does not even notices additional heirs. This offers a lot of options, such as separate delivery of ConcreteClasses or dynamic adaptation to the current context.

- *Type safety:* The idea of the pattern is to decouple the Client from the subsystem by weakening type checking. The parameter of the createElement method may be a basic type such as a string. The Factory interprets this string and evaluates the corresponding constructor parameters of the ConcretClass. Therefore, it is impossible to ensure that the parameter of the createElement method is correct. The string parameter may as well contain some corrupt data instead of a valid element identification. Hence, the Factory has to deal with faulty parameters too. Reasonable actions include throwing an exception or creating a default object. Still, the loss of type checking is a potential risk you should attack with thorough testing and code reviews.

Implementation Issues

- *Static Factories versus registration*. The easiest - and least reusable - way to implement the Factory is hard coding. The Factory just has a switch statement with a separate branch for every ConcreteClass. This solution has the least overhead and enables static type checking between the Factory and the ConcreteClasses. However, it requires the Factory to know all ConcreteClasses. Every new ConcreteClass causes a change of the Factory and it is impossible to add ConcreteClasses dynamically.

  Alternatively the ConcreteClasses register at the Factory during system startup. In this scenario the Factory is a dictionary with parameter values as key. The dictionary may contain prototypes of every ConcreteClass according to the Prototype pattern [GOF95]. In C++ the dictionary may even contain pointers to the corresponding constructors. The registry solution permits to add new ConcreteClasses dynamically during runtime. However, care has to be taken to correctly initialize the subsystem before the first access to the Abstract Interface. If a class fails to register then the system may generate exceptions instead of helping it's user to do her job.

- *How to retrieve possible values.* Since the Client does not know the ConcreteClasses it needs a list of permitted values for the parameter of createElement. With a static Factory, this list may be hard coded - a cheap but not very pleasing solution since every new ConcreteClass results in a change of the Client. A better way is to introduce a method giveValueList of the AbstractInterface, which delegates retrieval to the Factory.

- *Factory class versus class method.* Static factories are simple, so it might be a good choice to implement the corresponding code as a class method of the AbstractInterface. Beware with this solution: the superclass is statically coupled to all its heirs because it has to know every constructor. Adding a new ConcreteClass may even result in a recompile of the complete inheritance tree. Separating the Factory is the better design which adheres to the principle to separate concerns.

Sample Code

The C++ Code shown here implements a part of the text subsystem in the Example Resolved section. It uses an Abstract Factory, and a Prototype pattern to implement a factory to which arbitrary text elements can register. First we have a look at the subsystem's interface, the abstract Text class:


```
class Text
{
protected:
    // Standard constructors are protected to enforce
    // creation via createTextElement
    Text () {};
    Text ( const Text & aText ) {};

public:
    // public construction and deletion
    static  Text* createTextElement ( const string & aString
);
    virtual ~Text() {};

    // protocol for setting and retrieving data
    virtual const string& getText () const = 0;
    virtual void setText( const string &) = 0;

    // protcol for TextSubsystem administration
    static  void initialize ();
    static  void shutdown ();

    // protocol for the TextFactory
    virtual Text* clone ( const string & aString ) const = 0;
    virtual bool isValidString ( const string & ) const = 0;

};
```

The class hides its standard constructor and exports the static createTextElement method instead. The methods setText and getText store and retrieve text elements. Two methods clone and isValidString provide services for the factory as we shall see later. The createTextElement method delegates its task to the TextFactory:

```
Text* Text::createTextElement ( const string & aString)
{
    return TextFactory::getInstance()->
        createTextElement(aString);
};
```

Finally, there are two methods to initialize the subsystem before the first use and to do cleanup. Initialization means to register all text elements at the TextFactory:

```
void Text::initialize ()
{
    TextFactory * theFactory = TextFactory::getInstance();

    theFactory -> registrateAsDefault(new LiteralText());

    theFactory -> registrate(new CurrentDateText());
    theFactory -> registrate(new CurrentTimeText());
    theFactory -> registrate(new LiteralText());
    theFactory -> registrate(new CompoundText());
};
```

This is the only place in the system that has to know *all* text element classes. Therefore, it is a good idea, to define it in a file on its own and make sure, that no other files depend on it. There are three alternatives to this nasty solution:

1. Define a constant table with function pointers to every constructor. Let the initialize iterate through the table. When you add new elements, you have to build a new table and link it again but the changes concentrate on data rather than on code.

2. When your system supports dynamic link libraries (DLLs), it usually provides a defined initialization call. This is a good point to register all text elements contained in the DLL at the TextFactory. The system may retrieve the information about available DLLs from a configuration file. With this solution it is possible do add new text elements during run-time.

3. Declare a class init<textElement> for every text element. This class has an constructor that registers the text element at the TextFactory. Define an initialized variable at file scope of type. The idea is that the compiler initializes this variable at startup calling the constructor of init<textElement>. With this technique every class registers automatically. However, C++ does not guarantee that the variable is initialized at *startup* but „*before the first use of any function or object defined in that translation unit*.“ [Str+91, §3.4] So this technique might work with a certain compiler version but it is not safe. If you use dynamic linking, it does not work at all.

The registrateAsDefault method is new in this sample. It registers the prototype as a fallback decision if no other prototype matches. The declaration of TextFactory shows details:

```
class TextFactory
{
protected:
    list<pText> prototypes;
    pText       defaultPrototype;

    TextFactory ()
        : prototypes(), defaultPrototype ( NULL ) {};

public:
    static TextFactory* getInstance ();
    ~TextFactory();

    void registrate ( const pText aPrototype );
    void registrateAsDefault ( const pText aPrototype );

    pText createTextElement ( const string & aString ) const;

};
```

The class is a Singleton, therefore the constructor is protected. A list contains all prototypes that have registered until now. The defaultPrototype determines, which class the factory instantiates, when no other class matches. The registrate and registrateAsDefault methods are straight forward, so we shall only have a look at the createTextElement member function:


```
pText
TextFactory::createTextElement ( const string & aString )
const
{
    pText result = NULL;
    list<pText>::const_iterator protoIter;

    // Look for the first matching prototype;
    for (protoIter = prototypes.begin();
        protoIter != prototypes.end();
        protoIter++)
        {
            if ((*protoIter)->isValidString(aString))
            {
                result = (*protoIter)-> clone(aString);
                break;
            };
        };

    // if no prototype is found, generate a defaultPrototype
    if (result == NULL)
    {
        result = defaultPrototype -> clone(aString);
    };

    return result;
};
```

The method scans through the prototypes and asks every one whether it feels responsible for the particular string. There is no order between the prototypes, so you have no choice but to do a linear search. If the TextFactory finds a suitable prototype, it creates a new instance of that class using the clone method.

Variants

Multiple Factories to support different authorization levels. In most cases the Factory is a Singleton. Yet, there are systems where the set of available ConcreteClasses changes according to the context. If the system works at a certain authorization level, the user may not be allowed, to create every ConcreteClass. An elegant way to satisfy this requirement is to have different dictionaries for every authorization level, implemented as separate instances of the Factory.

Related Patterns

If the hierarchy is very large or changes frequently, you should consider a Reflective Architecture [BMR+96] instead. Describe the different heirs using a meta-model and meta-data. Implement the meta-model as subsystem and store the meta-data as configuration data. You may even start with an Abstract Interface and change to a Reflective Architecture in a later version, changing only the implementation of subsystem but not the clients.

The pattern is a combination of the Subsystem Facade (pp. 9) and the Exemplar Community Idiom [Cop92, Chapter 8.2]. While the Subsystem Facade ignores issues of object creation, the Factory Method is not intended for decoupling. The Abstract Factory corresponds to Late Creation Pattern [BRi96] and to the Abstract Constructor [Lan96], if the Factory is a registry.

The Factory usually is a Singleton [GOF95] and often uses the Prototype pattern [GOF95] to instantiate ConcreteClasses.

Known Uses

Jim Coplien presents this pattern as an example for the Exemplar Community Idiom [Cop92, Chapter 8.2].

The SIGMAPLAN project used this pattern to encapsulate configuration of message texts. The example sections of this pattern are driven from this project.

Inverted Association

Abstract

If you have a bi-directional association between two classes, a naive design results in a cyclic dependency. The Inverted Association uses an abstract class to break this cycle if the two directions are not balanced.

Example

Consider the detail of our Laboratory Management System depicted in Figure 11. The Instrument class does not send commands directly to the physical instrument. Instead it uses a lower level class InstrumentInterface, which is responsible for managing the communication with the physical instrument. The connections are bi-directional, because the Instrument not only issues commands but also reflects the status of the physical instrument.

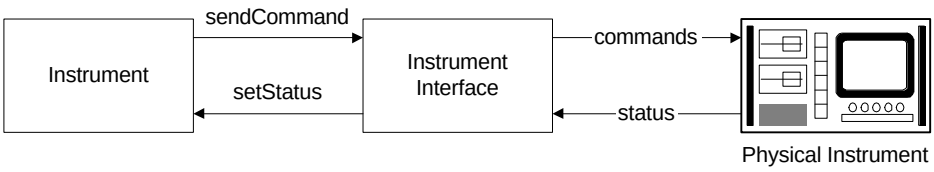


Figure 11: The communication between the Instrument class and the physical instrument

In the naive design of Figure 11 the InstrumentInterface directly calls the Instrument when it receives a new status from the physical instrument, resulting in a cyclic dependency between the two classes.

Context

A bi-directional association causes a cyclic dependency between two classes. The two directions are not balanced but you can identify one direction as the main path.

Problem

How do you break the cyclic dependency?

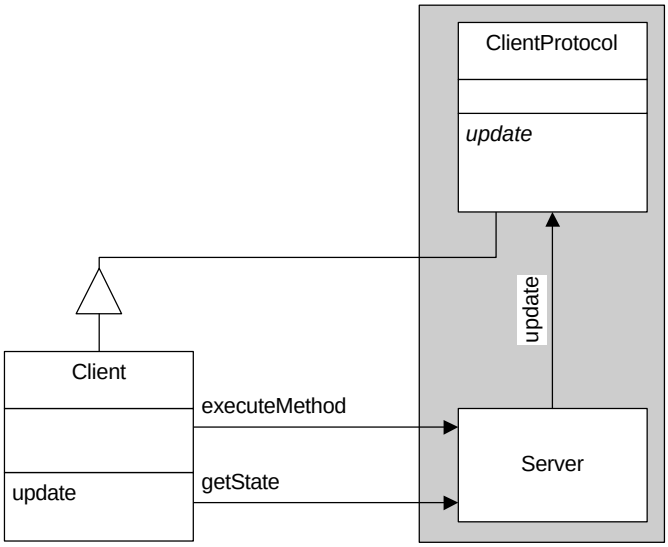
Forces

- *Acyclic dependencies versus type safety:* To develop the two classes independently, the dependency graph should be acyclic. However, the compiler needs both directions of the dependency to perform type-checking.
- *Communication complexity:* Communication between the subsystems should be as simple as possible. Complex communication makes the design harder to understand and impedes testing as well as debugging.
- *Extra Classes:* Every extra class raises the cost of the system in terms of money, time and complexity. Thus, if extra classes are unavoidable, they should at least be as generic as possible.

Solution

Apply the Dependency Inversion Principle [Mar96b]: Identify the main direction of the association. Make its origin the Client and its destination the Server. Define an abstract ClientProtocol class and put into the Server's subsystem. Add an abstract method to the ClientProtocol for every callback from the Server to the Client. Make callback every a call to the ClientProtocol. Make the Client heir of the ClientProtocol and redefine the abstract methods.

Structure



Participants

- **Client**
  - uses services the Server offers
  - monitors the state of the Server
- **Client Protocol**
  - belongs to the Server's subsystem
  - defines the methods the Client has to provide
- **Server**
  - offers services that are independent of the Client
  - class only the ClientProtocol for callbacks

Dynamic Behavior

The dynamic behavior has not changed much. The Client either instantiates the Server or connects to it, establishing the backward link in both cases. When the Server wants to call back the Client it calls methods of the ClientProtocol instead (Figure 12). Because the Client inherits these methods from the ClientProtocol the dynamics are nearly the same. However, coupling between Server and Client is now dynamic. The runtime system resolves the dependency using dynamic binding.

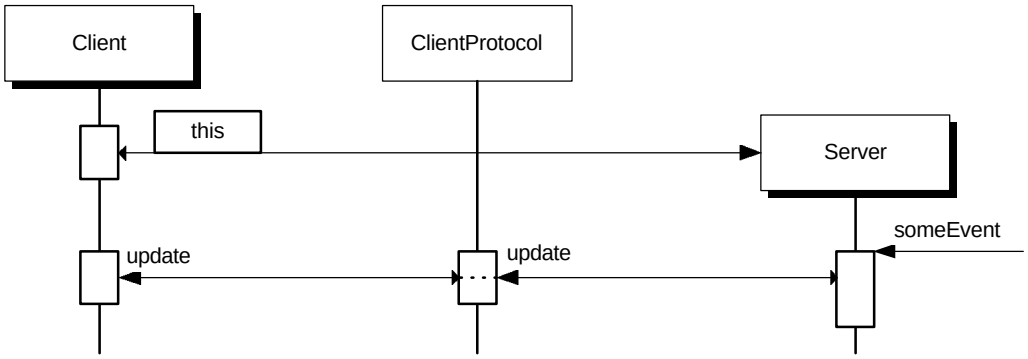


Figure 12: A Client instantiates a Server registering itself and issues a callback later using an Inverted Association. The Server calls only the abstract methods of the ClientProtocol. The runtime system resolves this call through dynamic binding (dashed line).

Example Resolved

The InstrumentInterface takes the role of the Server and the Instrument class forms the Client. You a new class InstrumentInterfaceProtocol to the Instrument Interface subsystem with a single, abstract method named statusChanged. The Instrument inherits this method from InstrumentInterfaceProtocol and redefines it with a call of getStatus. Figure 13 shows that the InstrumentInterface is unaware of the Instrument class now.

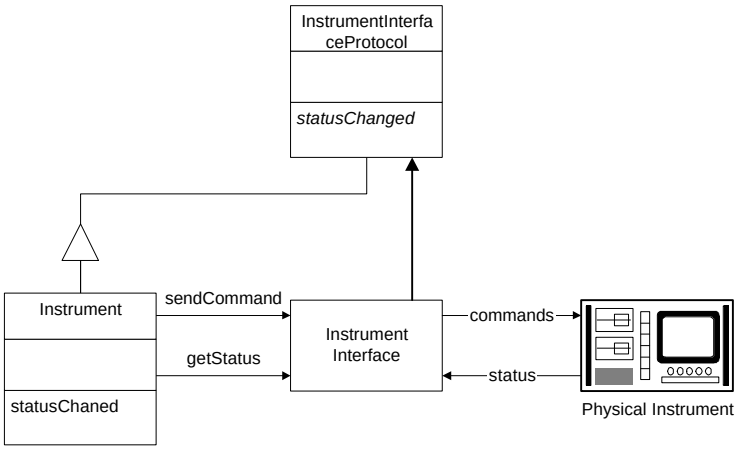


Figure 13: The Instrument Interface decoupled with an Inverted Association. There is a new class InstrumentInterfaceProtocol, which inverses the dependency between the Instrument and the Instrument Interface subsystem.

Consequences

- *Acyclic dependency:* The two classes depend acyclic upon each other. The resulting dependency graph contains no loops (Figure 14).

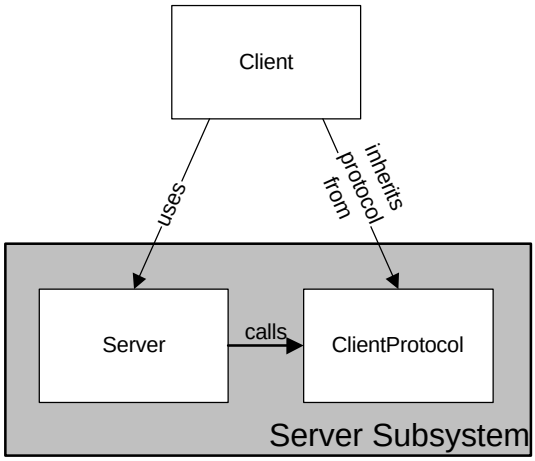


Figure 14: Resulting subsystem dependency graph.

- *Type safety:* The Client has to inherit the complete interface from the ClientProtocol, the Server will not work with any class that is not a heir of it. This ensures type safety of the design.
- *Communication complexity:* There is one additional level of indirection when a Server issues a callback. The runtime system hides this indirection with dynamic linking. Therefore, the code is as complex as before. During runtime there is small speed penalty you have to pay to resolve the dynamic binding.
- *Extra Classes:* ClientProtocol is specific for the particular Server, but not for the Client. Therefore the number of additional classes is bearable.
- *Reusability:* The Server subsystem becomes more re-usable. Because it has no notion of the Client, other classes may use the Server as well. However, this pattern allows only 1:n relationships between Client and Server. If a Server may have several Clients an Observer [GOF95] is more appropriate.

Implementation

- *Multiple Servers:* If a Client interfaces to several Servers, you may either use multiple inheritance or aggregate the corresponding ClientProtocols, depending on what your language supports. There is no problem here with multiple inheritance, because you only inherit interfaces. There is no danger to run into the diamond problem.

Sample Code

The following C++ code implements a minimalist version of the example. The InstrumentInterface class holds a status that may change ‘asynchronously’ when anyone calls the setStatus method. The class stores its reference to the client is the client member variable. Note, that the only constructor gets the client reference as parameter and the reference is declared const. Therefore, an instance is hard-wired to a certain client from construction till deletion. More flexible solutions need additional code.


```
class InstrumentInterface
{
protected:
    int status;
    InstrumentInterfaceProtocol* const client;

public:
    typedef int InstrumentStatus;

    InstrumentInterface (InstrumentInterfaceProtocol*
theClient,
        InstrumentStatus newStatus = 0);

    InstrumentStatus getStatus () const;
    void setStatus ( InstrumentStatus newStatus );
};
```

The only interesting method of the class is setStatus, which notifies its client:

```
void InstrumentInterface::setStatus (InstrumentStatus
newStatus)
{
    status = newStatus;
    assert (client != NULL );    // Check class invariant
    client->statusChanged();
};
```

Because the client attribute is only a pointer, you do not have to include the InstrumentInterfaceProtocol header. It is sufficient to put the following forward declaration in front of the InstrumentInterface declaration:

```
class InstrumentInterfaceProtocol;
```

Therefore the InstrumentInterface class needs no includes for its declaration, it has no other static dependencies.

The InstrumentInterfaceProtocol class provides a single, pure virtual function as protocol for all clients:

```
class InstrumentInterfaceProtocol
{
public:
    virtual void statusChanged() = 0;
};
```

The Instrument class inherits this protocol and redefines the statusChanged method:

```
class Instrument : public InstrumentInterfaceProtocol
{
protected:
    InstrumentInterface* myInstrumentInterface;
    bool    statusChangedFlag;

public:
...
    virtual void            statusChanged();
...

};
```


This declaration also shows the not-inverted reference to the InstrumentInterface.

Related Patterns

If the number of clients changes dynamically, you need a more complex solution, where clients register at their server. The appropriate pattern to use is the Observer [GOF95, p. 293] Every Observer may know several Subjects of different types. The Observer is more powerful but it involves more overhead in terms of complexity and ease of use.

Mediator [GOF95, p. 273] uses the Inverted Association to decouple the Mediator class from the Colleague class. Here a single client (the Mediator) uses several servers (the Colleagues) to do its work.

Association Object (see page 30) makes both directions of a cyclic dependency anonymous which results in even looser coupling. However, the resulting design is more complex and harder to use.

Known Uses

The Tools ++ library of Rouge Wave uses this pattern to emulate Smalltalk-like containers in C++. A class acting as items in a container has to inherit from a class RWCollectable, which provides the necessary interface.

The BTS DD35 Sidepanel project used this pattern to decouple special control elements, called “Digipots”. These controls were knobs on a video control panel, which had a indicator on the screen, displaying their current value. A class Digipot encapsulated the complete functionality. Clients of this class had to inherit from a class DigipotObserver to get messages when the turned the knob.

The Eiffel list described in [Mey88, chapter 9.1] also uses this pattern to decouple list from the listed items. A class LINKABLE provides the interface, listed classes have to provide in order to participate in a LIST.

Association Object

Also Known As

External Link Objects [Lin96], Association [Boy96]

Abstract

An Association Object completely decouples two associated objects by objectifying the association. It is a good way to solve cyclic dependencies, caused by bi-directional relations.

Example

Consider the bi-directional association between a Patient and her Orders depicted in Figure 15. A patient has assigned several orders which belong to her. Both directions are necessary for certain use cases. A physician may be interested in all the orders of a patient. Analogously, if an order has some problems, it may be necessary, to trace back to the patient.

Technically, you also need both directions. If the Patient class maintains a list of Orders, it has to update the list upon deletion of an Order. Hence, the Order has to inform the Patient in case of deletion.

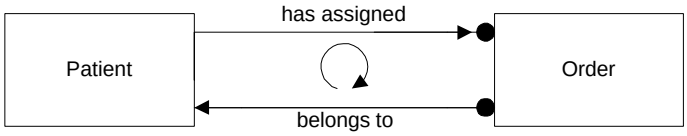


Figure 15: A cyclic dependency between Patient and Order

This cyclic dependency causes problems. Suppose, the Patient object is intended for reuse in a hospital management system. To administrate hospital beds, you need no orders. Thus, a Patient knowing Orders is less reusable.

Context

During design you encounter two classes that have a bi-directional association. They either belong to different subsystems or they are candidates for separate reuse. There is no main direction, both directions are equally important.

Problem

How do you break the cyclic dependency the association implies?

Forces

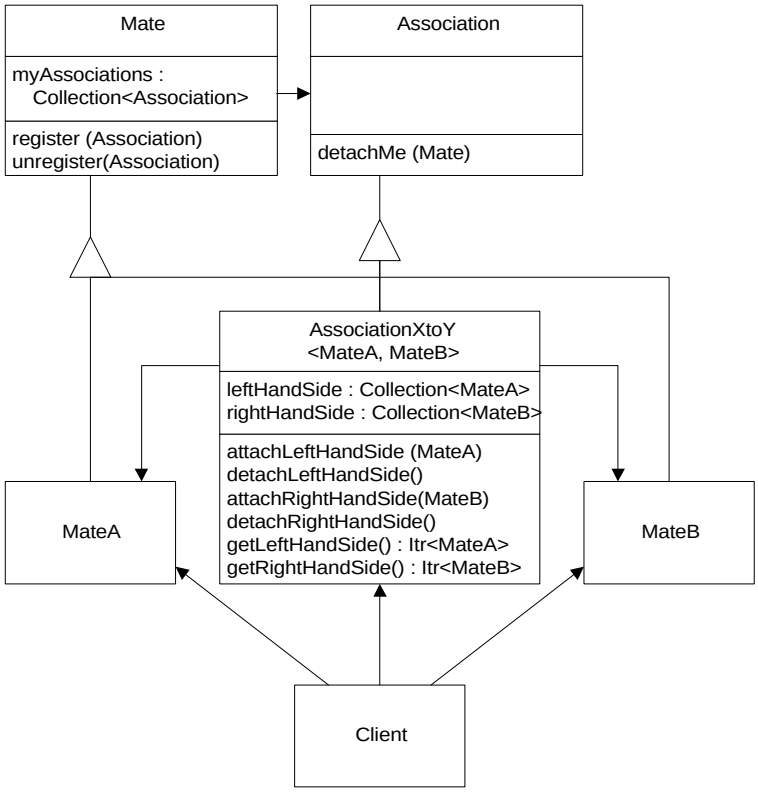
- *Decoupling versus complexity:* Without decoupling, you may have problems to implement and test the two classes separately. However, decoupling a balanced association may introduce significant overhead in terms of additional classes, design complexity, and communication complexity.

- *Reusability:* Any additional dependency makes a class more specific to its context. To maximize reusability, you should minimize dependencies to other classes that may not fit to a different context.

Solution

Introduce a separate *Association Object* to handle the association. Define abstract protocols to handle the communication between the classes and their associations.

Structure



Participants

- **Mate A**, **Mate B**
  - associated objects
- **Mate**
  - base class for all associated objects
  - detaches a Mate from all attached Associations in its destructor
- **Client**
  - owns and controls the association and the mates
- **Association**
  - base class of all associations
  - provides a protocol for the mates to detach themselves
- **AssociationXtoY <MateA, MateB>**
  - class template implementing the complete association management

- offers methods to retrieve the mates
- different versions exist for different multiplicities: Association1to1, Association1toN and AssociationNtoM
- May also contain domain level attributes of the association

Dynamic Behavior

Figure 16 shows the object interaction for a typical life cycle. The Client is responsible, to create all Mates and the Association object (❶) and to attach them to one another (❷). The Association registers itself at the Mate. The Client may detach one Mate explicitly (❸). If it deletes a Mate, it detaches itself automatically (❹).

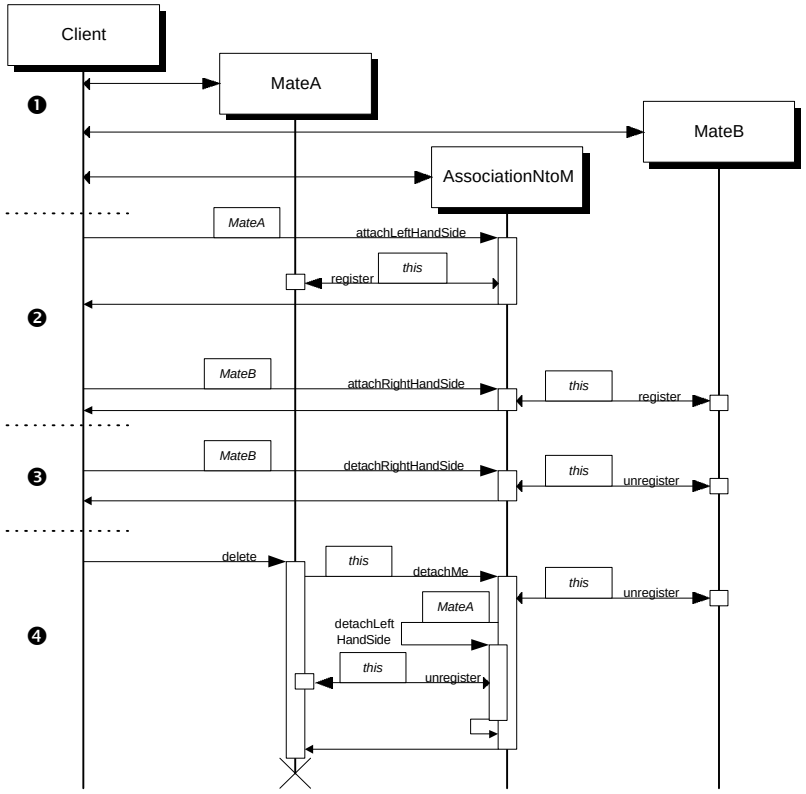


Figure 16: Collaboration of the classes. The scenario shows how all objects are created (❶), how mates are associated (❷) and the two possibilities to detach mates from an association (❸ and ❹).

Example Resolved

You can see the initial example after applying Association Object to it in Figure 17. Association1ToN knows both the Patient and the Order. To guarantee referential integrity, both Patient and Order send a detachMe message to an abstract Association class. They inherit this behavior from Mate. Since the Association is a generic class, there is no cyclic dependency.

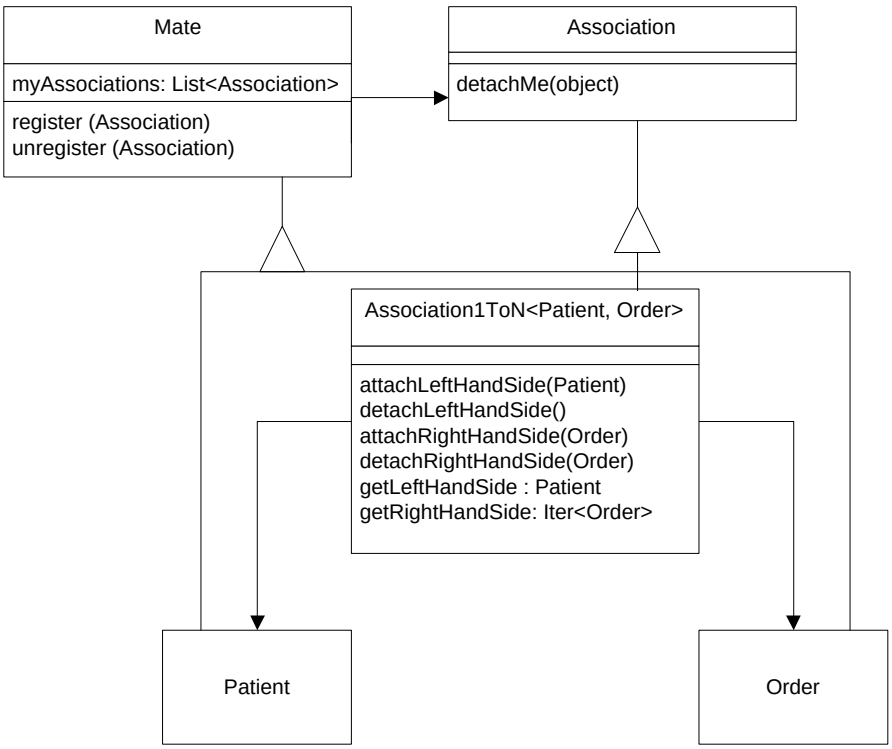


Figure 17: An Association Object between the Patient and the Order. The Patient knows only a generic Association object. A more specific heir models the Association itself and perhaps all attributes and methods that effect both sides of the association.

The object dependency graphs in Figure 18 show the effect of the new design. The cyclic dependency on the left hand side of the figure has moved from the domain level into the lowest level objects Mate and Association. Since both are generic, the structure now allows separate development of the higher level classes.

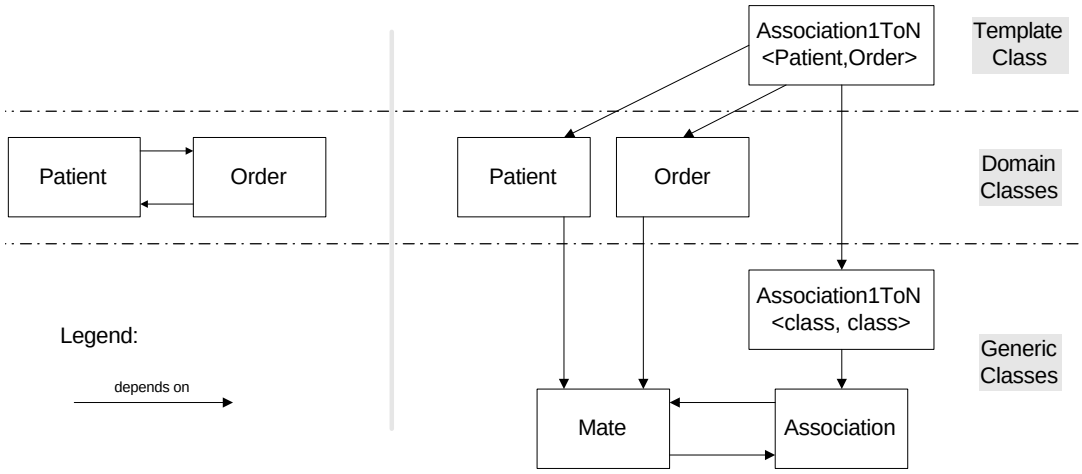


Figure 18: Object dependencies with the naive approach of Figure 15 on the left hand side and with the decoupled design of Figure 17 on the right hand side.

Consequences

- *Decoupling:* The pattern shifts the cyclic dependency from domain classes to generic classes. Thus it decouples the domain classes and enables the designer to put them into separate deliverables.
- *Complexity:* The design is more complex. There are additional generic classes with complex interaction. This makes the application harder to understand and harder to debug. If the additional classes are part of the basic services, this drawback is not as hard as it might seem.
- *Reusability:* The domain level objects are independently reusable. Since MateA and MateB are mutually unaware, they are reusable on their own.

Implementation Issues

- *Implement the pattern with an Observer mechanism*. The relation between the Association and the Mate conforms to the Observer pattern [GOF95]. The Association classes subscribe to the Mates for deletion messages. Hence the easiest way to implement the relation between both is to use an Observer mechanism from a class library.
- *Use the* Association *classes to hold information about the association.* Sometimes an association has additional attributes. An association between an employee and his company may contain the salary, the date of employment and so on. A good way to implement these properties and corresponding methods is to have them as attribute of the concrete association class as shown in Figure 19.

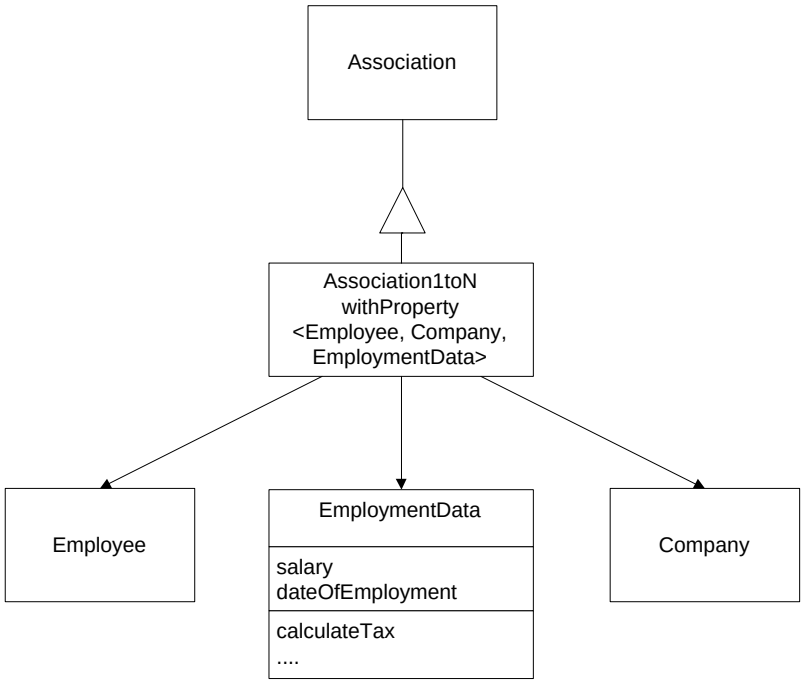


Figure 19: An association with properties and corresponding domain level methods

- *Use templates with C++*. Using class templates for the association classes ensures correct typing of all methods. It leads to completely generic classes that can be taken from a class library or a framework. If your compiler lacks support for templates, it is probably best, to use untyped references.

- *Use of multiple inheritance*. The inheritance relation between the Mate and the domain classes may lead to a multiple inheritance structure. This is bearable, because Mate only defines a protocol. However, if you are not able to use multiple inheritance, you may incorporate the protocol into a root class or sacrifice the automatic referential integrity.
- *Be careful with persistent* Mates. If one or both Mates are persistent, manipulations of the association should work without loading the corresponding objects into memory. If the implementation violates this rule, unnecessary database accesses might have negative impact on the performance of the system.

Variants

A simpler implementation of this pattern works without automatic referential integrity. In this case the Association and the Mate base classes are unnecessary. The Client has to guarantee referential integrity by calling the detach methods of the Association whenever it deletes a Mate object. This solution simplifies the design of the association. However, it results in a more complex Client. So, before choosing this variant you should carefully analyze whether it really simplifies the system or just moves complexity from generic classes to domain classes. The latter is seldom a good idea.

Sometimes the single Association object is split into two separate classes, one for the left hand side and one for the right hand side [Lin96]. These two objects know each other and maintain the association between them using a standardized protocol. There are two different association classes, one for a single client and one for multiple clients. This solution offers the possibility to combine the different association classes forming 1-to-1, 1-to-many or many-to-many associations. On the other hand it implies more classes that are more difficult to handle.

Related Patterns

The Mediator pattern [GOF95] is similar in structure and intent. Still, the Mediator decouples a highly interwoven set of classes while the Association Object decouples two classes with domain level associations. The Association Object may be interpreted as a special variant of the Mediator.

Known Uses

Christian Tanzer sketches this pattern in a fundamental discussion of associations [Tan95].

The Hypo project at sd&m used this pattern to transform foreign key relations of an RDBMS into the object model [KMW96].

Terris Linenbach describes a set of C++ classes that use this pattern to implement associations [Lin96]. The article outlines a variant without the Observer mechanism and with a pair of association classes forming an association. Two association classes form a single association. Every association class has one link to a domain level object and a second link to another association object.

Director

Abstract

As you add more and more use cases to a systems, the coupling of classes often increases. The Director provides a technique to cope with this form of coupling objectifying the control of use cases.

Example

Consider canceling an order in the laboratory management system. The laboratory attendant may have dropped a sample, the physician may have written an erroneous request, or the sample turned out to be polluted, to mention some of the possible triggers for this use case. Canceling an Order is not a trivial task:

- You also have to cancel all corresponding TestRequests and tag their TestResults as canceled.
- You have to check the status of the attached Samples. If they are not yet processed, you may want to delete them, especially when the test is expensive.
- The Requester has to get a confirmation that her Order has been canceled.

A first-cut design attaches methods to the existing objects calling each other (Figure 20). Since there are more than hundred similar use cases, there is a high probability for cyclic dependencies and ‘Hyperspaghetti objects’.

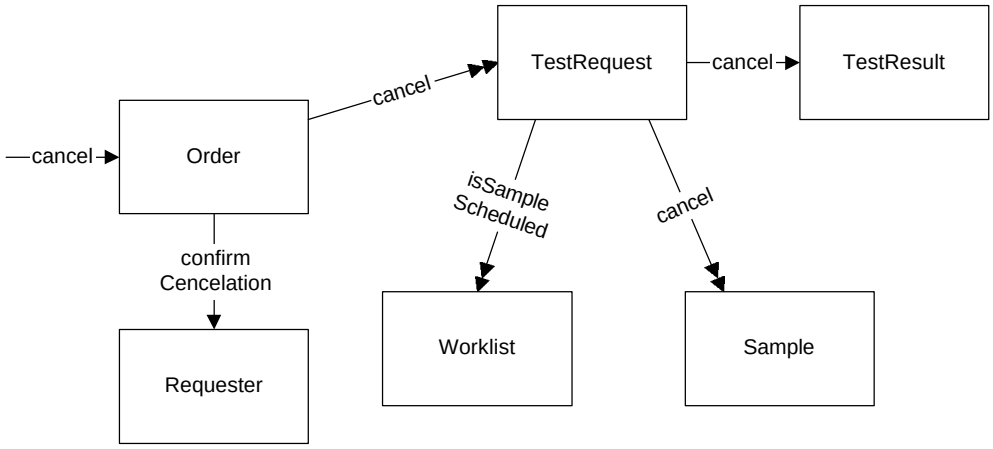


Figure 20: A naive design to cancel an Order. Since other use cases will result in similar call sequences, the Objects will get more and more coupled with each new use case. Finally you will end with ‘Hyperspaghetti objects’.

Context

You are designing a large system with many complex use cases. Many use cases touch several objects resulting in new dependencies. The closure of all these dependencies is a tight web.

Problem

How do you reduce dependencies between the objects?

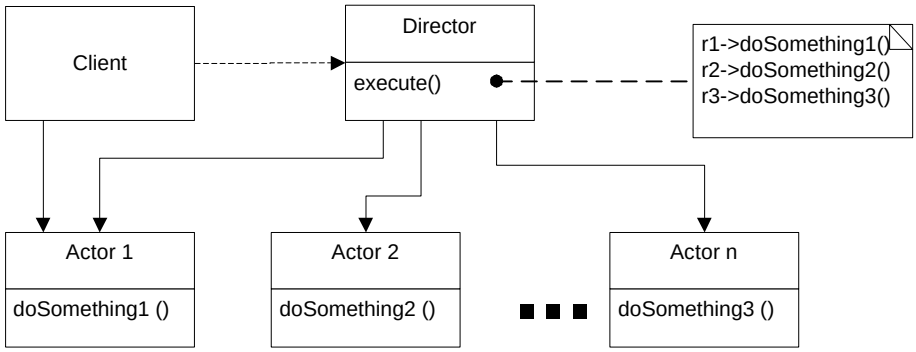
Forces

- *Decoupling:* The tight coupling of the classes makes separate development hard. Especially if you couple all domain classes of the system this may be a major problem in a large project.
- *Representational approach versus maintainability:* The dependencies between the objects stem from the different use cases every object supports.[7] Separating use cases from the objects tends to result in a functional design on a data model - a Representational Approach [Mar95]. However, functional designs do not exploit commonalties between different functions and therefore seldom lead to good maintainability.

Solution

Introduce a new Director class for every use case. Limit calls between the other classes to activities you find in many use cases. Let the Director control the other inter-class calls.

Structure



Participants

- **Director**
  - Has references to every Actor, involved in the use case.
  - Has (at least) one method execute, responsible to call all methods of the Actors in an appropriate order. This method also manages the parameters, passed between the objects.
  - Is the only class, a client may call, to perform the desired action.
  - Usually lives very short-termed
- **Client**
  - Creates the Director and provides one or several references to Actors
- **Actor *n***

[7] You may also consider the problem as methodological. Database applications such as business information systems are structured along the data abstractions and not along behavioral aspects. If you just add the use cases to every representation, like we did in the example, your system heavily depends on the concrete representation and is not open for extensions anymore. This is the "real" problem we have here [Mar95, pp. 283].

- Represents the objects the use case works upon
- Does ‘local consistency checks’ on its attributes
- Lives for many use cases
- Provides methods needed for several use cases
- Lacks methods needed only for a single use case implemented by the Director

Dynamic Behavior

- The Client creates the Director, supplies a reference to at least one Actor and starts execution of the Director.
- The Director might find additional references to Actors and calls all the methods of the Actors. It may call the methods in a sequence or adhering to a complex control flow. The Director is also responsible for passing parameters from one method to the other.
- The Actors perform tasks that are local to their attributes or tasks that needed for several use cases.
- The Actors also detect errors and signal them to the Director. It’s the responsibility of the Director to choose a proper reaction.

The following diagram demonstrates the collaboration:

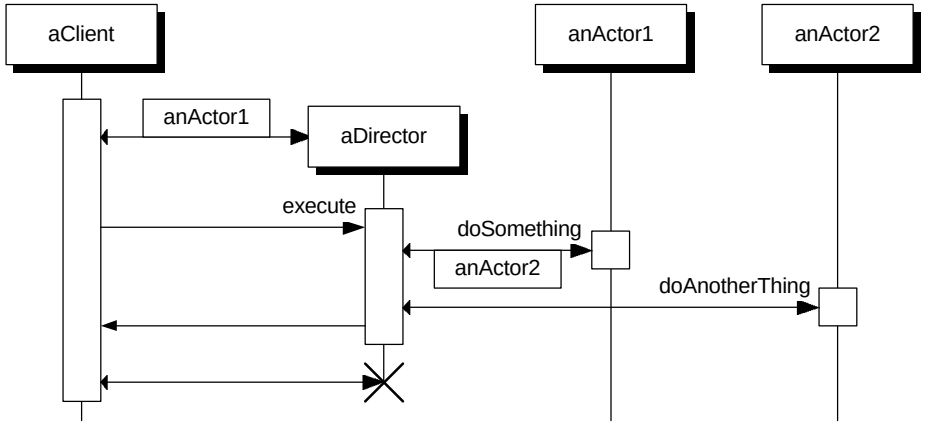


Figure 21: A sample interaction diagram for the Director pattern.

The Client instantiates a Director and supplies at least one Actor[8]. Later, it calls the execute method of the Director which starts to call the Actor. Perhaps the first Actor returns a reference to another Actor which the Director has to call. The execution of the use case may be arbitrarily complex. Finally, the Client deletes the Director.

Example Resolved

Figure 22 shows the CancelOrderDirector of the laboratory manager. The irregular relations of the old approach have disappeared and the design now has a layered structure: The CancelOrderDirector knows all the other classes but not vice versa. There is no risk of cyclic dependencies.

8 The Client still sends a message to an object, so it has to specify the final receiver of this message.

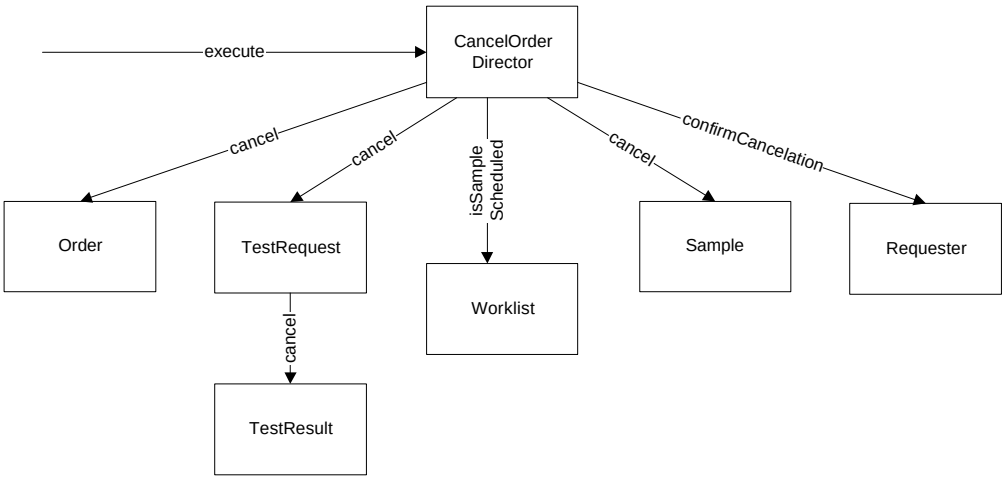


Figure 22: A Director decouples the classes.

Consequences

- *Good decoupling:* The pattern is a very powerful way to decouple Actors from each other. It eliminates references only needed for a single use case, and structures them. This is especially useful for long-term maintenance, since new use cases only add new Directors, but no new complexity. The Directors need much knowledge about the Actors but not vice versa so we have a layered design.

- *Representational approach and maintainability:* The frequent use of this pattern tends to produce a "representational structure" of the Actors [Mar95]: most of the functionality is located in the Directors while the data is stored in the Actors. In extreme, the design exists of data classes and functions. To preserve maintainability it is a good practice is to place methods, needed for more than a single use case, into the appropriate Actors. These methods are usually more general in nature, and are therefore good candidates for reuse.

Implementation Issues

- *Director as construction rule:* You may use the Director to cope with a single hot-spot as well as a general construction rule for a certain part of the architecture. While the former is mainly straight forward, care has to be taken with the latter. Avoid to completely separate data and functionality. Having dumb data objects and hyper-intelligent directors just moves the chaos from the Actors to the directors without any gain. Neither the Actors nor the Directors would be reusable with this design. However, a sensible usage of this pattern may enhance design clarity and reusability.

- *Director and transactions.* If you use this pattern as a general design rule, the Directors are a good guess to place transaction brackets.

- *Finding the Actors:* Since the Director works with many different Actors, it is quite an interesting issue, how it finds the correct objects. When there are few Actors, the Client may supply the references. As the number of Actors increases, this way causes tighter and tighter coupling between the Client and Actors. A frequent solution is to pass only a root object to the Director, while the execution of the use case is split into two steps: During the

first phase the Director collects all the Actors, the second phase serves to accomplish the task.

- *Nesting Directors:* For a complex use case, it may be appealing, to nest Directors: A higher level Director triggers lower level Directors, which, in turn, trigger the Actors. This structure may avoid redundancy, but it also has a pitfall: if the Director has an internal state machine, you have to ensure, that the life cycle makes sense for nested calls *and* for direct calls. As an example consider a Director controlling transactions. Nesting Directors means nesting transactions. Most databases do not support this.

- *Director as Singleton*: Some languages, such as C++, put a certain runtime penalty on the instantiation of objects. If the design relies on Directors heavily there may be a significant time overhead for initializing and destructing Command Class objects. To avoid this the Director classes may be Singletons [GOF95].

Variants

Some systems have complex use cases, which live for several hours or even days. You may use a *Persistent Director* to cope with these requirements.

Related Patterns

The Director usually is a short-living class. The Client instantiates it to start a use case and deletes it as soon as the use case is done. Still, more complex models are possible. Directors may live on after the use case is executed to provide history or even undo operations. In this case, the Director implements a Command pattern [GOF95].

Visitor also encapsulates functionality. However, its intent is to implement one of several operations, that have to be performed on the complete object structure. Report generation is a good example. One of the main features of the Visitor pattern is the exchangeability of operations. So you may have different Visitors to generate different styles of reports, without changing the Actors. You may consider Visitor as an extension of the Director pattern.

Strategy is another design pattern to cope with functionality. It intents to make algorithms interchangeable. So it does not *control* the execution of an algorithm, but it *is* the algorithm.

The Electronic Clerk metaphor [Den91] is similar to the Director. However, a single Electronic Clerk may handle several different orders, not only one.

Known Uses

The Transaction Objects described in the payroll application in [Mar95] is an application of this pattern.

The EASY project of sd&m used persistent Directors to model complex business use cases. The use cases had a duration of several days and contained a number of transactions.

Acknowledgments

Thomas Belzner has shepherded this document and gave valuable hints for improvement. Andreas Mieth reviewed an earlier version and suggested further known uses. Thomas Kunst suggested to add the Inverted Association pattern. Last but not least Klaus Renzel and Wolfgang Keller suggested improvements in several review circles. Thanks to you all.

References

[BMR+96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal**: *Pattern-Oriented Software Architecture - A System of Patterns;* John Wiley & Sons Ltd., Chichester, England, 1996, ISBN 0-471-95869-7

[Boy96] **Lorraine L. Boyd:** *Patterns of Association Objects in Large Scale Business Systems;* EuroPLoP 96 - Preliminary Conference Proceedings, 1996; Siemens AG, München

[BRi96] **Dirk Bäumer, Dirk Riehle:** *Late Creation - A Creational Pattern;* PLoP'96 - Proceedings; Section 5.1; University of Illinois at Urbana-Champaign, 1996

[ChK91] **Shyam R. Chidamber, Chris F. Kemerer**: *Towards a Metrics Suite for Object Oriented Design;* OOPSLA'91 Proceedings, pp. 197-211, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991, ISBN 0-0201-55417-8

[Cop92] **James O. Coplien:** *Advanced C++ - Programming Styles and Idioms;* Addison-Wesley Publishing Company, Reading, Massachusetts, 1992; ISBN 0-201-54855-0

[Den91] **Ernst Denert**: *Software-Engineering - Methodische Projektabwicklung;* Springer-Verlag Berlin Heidelberg New York; 1991; ISBN 3-540-53404-0

[Gam92] **Erich Gamma**: Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge; Springer-Verlag Berlin Heidelberg New York, 1992.

[GOF95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley, 1995.

[HMo96] **Martin Hitz, Behzad Montazeri**: *Measuring Coupling and Cohesion In Object-Oriented Systems;* Institut für Angewandte Informatik und Systemanalyse, University of Vienna, 1996.

[KMW96] **Wolfgang Keller, Christian Mitterbauer, Klaus Wagner**: *Objektorientierte Datenintegration über mehrere Technologiegenerationen;* Proceedings ONLINE, Kongress VI; ONLINE GmbH, 1996

[Lan96] **Manfred Lange**: *Abstract Constructor;* EuroPLoP 96 - Preliminary Conference Proceedings; Siemens AG, München, 1996

[Lin96] **Terris Linenbach**: *Implementing Reusable Binary Associations in C++;* C++ Report, May 1996, pp. 40-51; SIGS Publications, Inc, New York,NY, ISSN 1040-6042

[Mar95] **Robert C. Martin:** *Designing Object Oriented Applications in C++ using the Booch Method*, Prentice Hall International, London, 1995, ISBN 0-13-203837-4

[Mar95b] **Robert C. Martin:** *Principles of OOD*; 1995; Available via http://www.oma.com/Offerings/catalog.html

[Mar96] **Robert C. Martin:** *The Open-Closed Principle - Design Models That Never Change And Yet Extend Their Behavior;* C++ Report, Jan 96, p. 37-43; SIGS Publications, Inc, New York,NY; ISSN 1040-6042

[Mar96a] **Robert C. Martin:** *The Liskov Substitution Principle*; C++ Report, March 96, p. 14-23; SIGS Publications, Inc, New York,NY; ISSN 1040-6042

[Mar96b] **Robert C. Martin:** *The Dependency Inversion Principle;* C++ Report, June 96, p. 61- 65; SIGS Publications, Inc, New York,NY; ISSN 1040-6042

[Mar96c] **Robert C. Martin:** *The Interface Segregation Principle - One of the Many Principles of OOD;* C++ Report, August 96; SIGS Publications, Inc, New York,NY; ISSN 1040-6042

[Mey88] **Bertrand Meyer**: *Object-Oriented Software Construction;* Prentice-Hall International, London, 1988, ISBN 0-13-629049-3

[Mey92] **Scott Meyers**: *Effective C++ - 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1992, ISBN 0-201-56364-9

[Par72] **D. L. Parnas:** *On the criteria to be used in decomposing systems into modules;* In Communications of the ACM, 15.12 (1972), pp. 1053-1058, ACM

[Sou94] **Jiri Soukup**: *Taming C++ - Pattern Classes and Persistence for Large Projects;* Addison-Wesley Publishing Company, Reading, Massachusetts; 1994; ISBN 0-201-52826-6

[Str+91] **Margaret A. Ellis, Bjarne Stroustrup:** *The Annotated C++ Reference Manual;* Addison-Wesley Publishing Company, Reading, Massachusetts, 1991; ISBN 0-201-51459-1

[Tan95] **Christian Tanzer**: *Remarks on Object-Oriented Modeling of Associations*; Journal of Object Oriented Programming, Februrary 1995, pp. 43-46; SIGS Publications, Inc, New York,NY; ISSN 0896-8438

[Web95] **Bruce F. Webster**: *Pitfalls of Object-Oriented Development;* M&T Books, New York, 1995, ISBN 1-55851-397-3

Document History

Version	Changes	Inspected by
0.1, 01.08.96	newly written	Andreas Mieth, 10.09.96 Th. Belzner, 24.09.96
0.2, 25.09.96	• rewrote ‘Basic Thoughts’ to define ambiguous terms • reworked the other patterns according to the detailed annotations	Th. Kunst, 11.11.96 Th. Belzner, 28.11.96 (released document)
1.0, 29.11.96	• Adapted to ARCUS pattern structure • Deleted rambling passages in the introduction and rewrote parts of it • Replaced Unilateral Relation by Inverted Association • Inserted Sample Code section	J. Siedersleben (Introduction) 18.11.96