Design Patterns, A Quick Introduction

Thomas Panas, Vaxjo University
Thomas.Panas@msi.vxu.se

September 2001

1 Introduction

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander (1977)

This article is a summarised insight into the field of design patterns. With the help of four books, namely Entwurfsmuster [1] (English: Design Patterns [2]), Java Design Patterns [3], Patterns in Java [4], Entwurfsmuster anwenden [5] (English: Pattern hatching [6]), the content of this paper was established.

This article is divided into mainly three different chapters. Chapter Two and Three give a basic insight into the area of design patterns, explain what patterns are for and how they can be found and used. You will not find a complete pattern catalogue here, rather one huge example that shows you how to use specific patterns for specific problems. Therefore Chapter Four introduces you to a practical problem that is being solved with the help of three design patterns. Those patterns are explained in detail whenever they appear.

The example is just a summarised part of a bigger example taken from the Entwurfsmuster anwenden book [5]. The design patterns explained during this example are taken out of all four books. They combine a comprehensive knowledge of all books but do not show an explicit example, since the example is given by the whole problem stated in Chapter Four. Chapter Two and Three are also a collection of knowledge from all of those books mentioned above.

We hope that this paper will give you a good starting point on how to read, learn, find and use design patterns. After reading the whole example, you should be motivated to continue reading all the other design patterns that are waiting out there to be investigated. And who knows, maybe you will write even your own design pattern after all!

2 Design Patterns

Software patterns are reusable solutions to recurring problems that occur during software development. Software patterns are made by experienced programmers and allow every other less experienced programmer to act as an expert. Experience gives programmers a variety of wisdom. As programmers gain experience, they recognize the similarity of new problems to problems they have solved before. With even more experience, they recognize that solutions for similar problems follow recurring patterns. With knowledge of these patterns, experienced programmers recognize the situations to which patterns apply and immediately use the solution without having to stop, analyze the problem, and then pose possible strategies.

When a programmer discovers a pattern, it is just an insight. In most cases, to go from an unverbalized insight to a well-thought-out idea that the programmer can clearly articulate is surprisingly difficult. It is also an extremely valuable step. When we understand a pattern well enough to put it into words, we are able to intelligently combine it with other patterns.

To understand and wisely use design patterns we must however fist go through the pain of reading and learning the patterns. Learning design patterns is a multiple step process. First, you accept the premise that design patterns are important in your work. Then, you recognize that you need to read about design patterns in order to know when you might use them. Finally, you internalize the patterns in sufficient detail that you know which ones might help you solve a given problem.

The legend book "Design Pattern" by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides merges patterns with similar structure into similar categories. These categories or classifications are in their book creational, structural and behavioral patterns. Other books as the Patterns in Java from Mark Grand add new patterns to those classifications or create completely new classifications. There is an infinite amount of areas where patterns can be used. Therefore there is also a wide classification possible in which design patterns can be set into. Here we want just give a small example of classifications, their purpose and patterns that fall in their category:

- *Creational patterns:* create objects for you, rather than you having to instantiate objects directly. Your program gains more flexibility in deciding which objects need to be created for a given case. Patterns that fall in this category are: Factory Method, Abstract Factory, Builder, Prototype, Singleton and Object Pool.

- *Structural patterns:* help you to compose groups of objects into larger structures, such as complex user interfaces and accounting data. Patterns that fall in this category are: Adaptor, Iterator, Bridge, Faade, Flyweight, Dynamic Linkage, Virtual Proxy, Decorator, Cache Management.

- *Behavioral patterns:* help you to define the communication between objects in your system and how the flow is controlled in a complex program. Patterns that fall in this category are: Chain of Responsibility, Command, Little Language, Mediator, Snapshot, Observer, State, Null Object, Strategy, Template Method, Visitor.
- *Fundamental patterns:* are the most fundamental and important patterns to know. You will find these patterns extensively in other design patterns. Patterns that fall in this category are: Delegation, Interface, Immutable, Marker Interface, Proxy.
- *Partitioning patterns:* In the analysis stage, you examine a problem to identify the actors, concepts, requirements, and their relationships that constitute the problem. Partitioning patterns provide guidance on how to partition complex actors and concepts into multiple classes. Patterns that fall in this category are: Layered Initialization, Filter, Composite.
- *Concurrency patterns:* involve coordinating concurrent operations. Patterns that fall in this category are: Single Threaded Execution, Guarded Suspension, Balking, Scheduler, Read/Write Lock, Producer-Consumer, Two-Phase Termination.

3 How to find the right pattern

Developing a program is not easy. No educated software developer starts nowadays just with hacking some code. What is needed is a picture of the "organic whole". There are mainly two ways to receive this picture that we can call a design. We can either sit down, think about the problem, discuss it with colleagues, build the solution up step by step and finally create a design as a long and well thought out thinking process, or we let others think for us. The second idea sounds good, or? The trick behind this idea is to reuse the knowledge of experts and we got this knowledge already. It is hidden in well formulated design patterns. All we have to do is to study and understand those patterns, so that we can really reuse them. The result is that we can design faster and qualitatively better, since we use expert knowledge.

But even when we are willing to read a design pattern book or search for a specific design pattern for our current problem, how can we be sure that we will find a pattern? Actually, we can not be sure that we will find one. But knowing how to search for a pattern gives us a higher chance of finding an appropriate pattern for our problem. This section will show you how to find the right design pattern.

Currently we face many kind of patterns, like design patterns, analysis patterns, organizational patterns etc. In this article we cope only design patterns. The three books discussed here: Java Design Patterns and Patterns in Java cover around 41 design patterns, pattern hatching adds even some more to it. Dealing with so many design patterns, it is of course difficult to find the right pattern. It is even more difficult when you are not familiar with the catalog of patterns. Here come some approaches on how to find a proper design pattern for your problem:

- *Rethink, on how design patterns solve design problems.* Design pattern helps you to find the right object, to find the object granularity, to define the interface of objects and so on. Read the pattern description and decide if the help a pattern gives is appropriate to your problem.
- *Cross-read the purpose sections. Each pattern has a purpose.* Read the purpose section of the patterns to find one or more patterns that could be relevant to solve your problem. Use the classifications that patterns are separated into. Patterns might be classified into structural, behavioral and creational patterns, but also into concurrency, partitional or fundamental patterns etc.
- *Observe how the patterns are related to each other.* The intensive observation of the relationships between patterns might help you to find the way to the right pattern.
- *Investigate patterns with the same task.* Design patterns are classified into different groups. Depending on the book you read the classification might be different. The classical design pattern book from the Gang of Four separates patterns into creational, structural and behavioral patterns. Each classification in a catalog ends with a comparison of commonalties between those patterns. This might help you to understand their specific task in their classification better.
- *Discover the reasons for design revisions.* Observe the reasons for design revisions in order to see if your problem imbeds one or more of those reasons. Observe afterwards the patterns that can help you to avoid those reasons for design revisions.

4 How to use the right pattern

Lets say that you have found the right design pattern and you are quite sure that this is exactly the one that you want to implement. And now you have the pattern in front of you together with your problem, but how to you implement the pattern to your problem to get a clean solution? The next part shows a step-by-step approach on how to implement a pattern:

1. Read the complete pattern once to get an overview. Devote the applicability and consequence part here special attention to be sure that the pattern is really the right one for your problem.
2. Go back and study once more the structure-, participant- and interaction-section. Be sure that you understand the classes and objects in the pattern and their relationships.
3. Take a look on the example code to see a concrete example of a pattern as source-code. The intensive observation of the code might ease you the implementation of a pattern.
4. Choose useful names for the objects and classes that participate on the pattern. The names given for object and classes in the pattern catalog are mostly very abstract and should be changed to context-oriented names.

5. Define your classes. Define your interfaces, establish your inheritance relationships and define your exemplar variables, which depict the data and object references.
6. Define appliance specific names for operations in the pattern. Hereby the name depend again on the context of the application.
7. Implement the operations so that they transact the cognizance and interaction according to the pattern. Take again a look on the examples that are provided in the pattern catalogs.

These aspects depict only aid for you to overcome the beginning problems of finding the right pattern. But studying some design pattern books will definitely give you a deeper understanding of each of the patterns and a following implementation gives you a feeling for how to work practically with these design patterns. Therefore this article continues with a huge example on how to design and implement a problem definition with the help of patterns.

5 Designing with Patterns

The best way to get a feeling for the use of design patterns is simply to use them. In this manner we will study a hierarchical file-system and try to implement patterns wherever possible. Actually, we will build up an entire design for a file-system from scratch, where we create the programming model, which application programmers use, the so called, Application Programming Interface (API). While developing this file-system step by step we search for suitable patterns that describe parts of our system and try to implement them. Through this example the practical use of design patterns shall be demonstrated. However, the idea with this is not to show the best way of how to design a hierarchical file-system, the idea is to show how to use the design pattern catalog in practice and encourage the reader to continue using patterns in all his/her future designs.

5.1 Basic file-system

From the view of the computer user, a file-system should be able to cope with data-structures of arbitrary size and complexity. There should not be a licentious limit on how huge or deep a data-structure could be. From the view of the programmer, it should be easy to work with such a data-structure and at the same time the data-structure should be easily extendable.

Assuming that you are implementing a command that lists the files in a directory. The code that you write to acquire the name of a directory should not differ from the code that acquires the name of a file. With other words, you should be able to tread directories and files in the same way when acquiring their names. The code resulting from this is easier to write and easier to maintain. Considering this we get a file-system structure such as in figure 1.

How do we implement such a structure? The fact that we have two kinds of objects here, directories and files, seduces us to create a class for each of them. But on the other hand we would like to treat them equally. This means they should have a common interface that we will call node from now on.

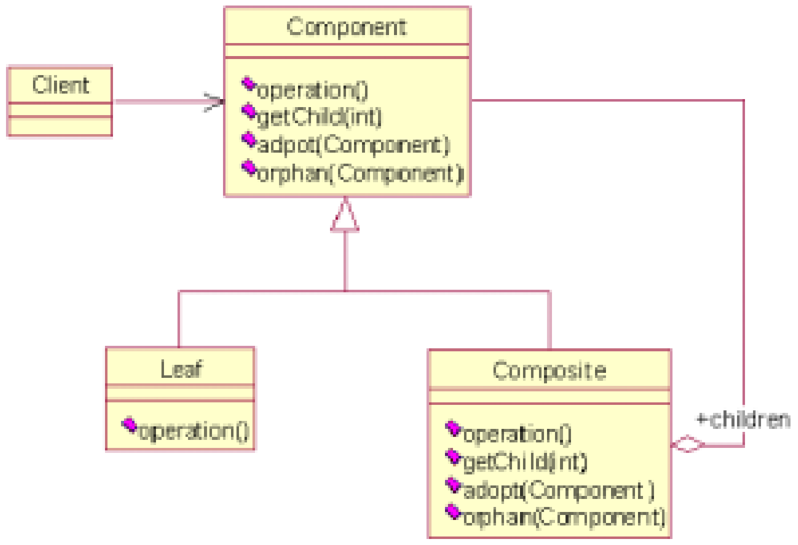

Figure 1: The file-system

All these regards lead us to the UML diagram of figure 2.

We can see an interface called node and two inherited classes file and directory. Both, the file and directory class implement some specific operations to their classes. Moreover, the directory class aggregates back to the node class. This means that each directory can itself contain some files and/or directories on its own. _nodes is the private member in our example that remembers all the children of this current directory. In order to access a child, we furthermore implement a function called getChild. This function must be implemented in both, the directory and the node class. This is because we want to be able to return both, files and directories with the getChild function. Considering this, the getChild function has to return a node object and not a directory or file object and therefore we implement the function getChild in both classes, directory and node.

Finally we got a basic structure of our file-system. The next step is to find a pattern that matches our design and try to implement it. If we look through all design patterns that we have, we will find a pattern called composite pattern that matches exactly our demands. This pattern is explained in the next section in depth.

5.2 Composite pattern

5.2.1 Description

The composite pattern is also known as the recursive composition pattern. It allows you to build complex objects recursively composing similar objects in a treelike manner. The composite pattern also allows the objects in the tree to be manipulated in a consistent manner, by requiring all of the objects in the tree to have a common superclass or interface.

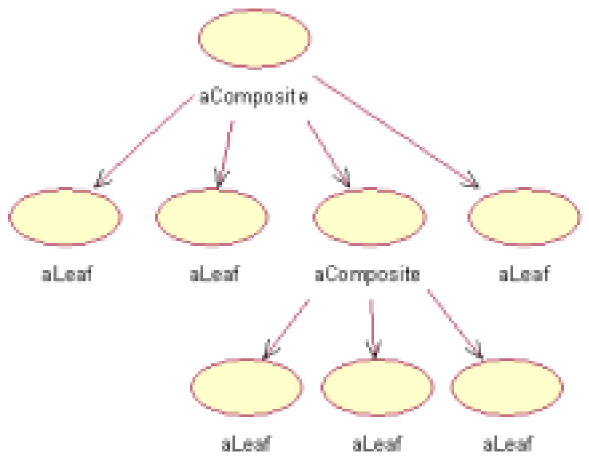

Figure 2: Basic UML diagram

5.2.2 Forces

- You have a complex object that you want to decompose into a part-whole hierarchy of objects.
- You want to minimize the complexity of the part-whole hierarchy by minimizing the number of different kinds of child objects that objects in the tree need to be aware of.

5.2.3 Structure

Figure 3 shows the basic structure of the composite pattern.

A typical object-structure of the composite pattern could look as depicted in figure 4.

5.2.4 Participants

- Component
  - declares the interface for objects in the jointed structure.
  - implements a default behavior for the interface
  - declares a interface to access and manage child object-components.
  - declares optionally an interface to access the parent object of a component within a recursive structure
- Leaf
  - represents child objects in the composition. A leaf does not contain child objects.

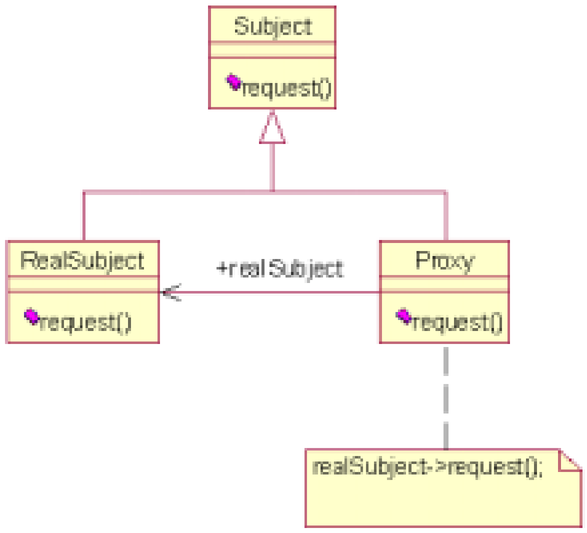

Figure 3: Composite Pattern

- defines a behavior for the primitive objects in the composition.

- Composite
  - defines the behavior for components that can contain child objects.
  - stores child object components.
  - implements child object related operations of the interface of component.
- Client
  - manipulates the objects of the composition through the interface of component

5.2.5 Consequences

The composite pattern allows you to define a class of hierarchy of simple objects and more complex composite objects so that they appear to be the same to the client program. Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way. The composite pattern also makes it easy for you to add new kinds of components to your collection, as long as they support a similar programming interface. On the other hand, this has the disadvantage of making your system overly general. You might find it harder to restrict certain classes, where this would normally be desirable.

5.2.6 Implementation

The intent of the composite pattern is to allow you to construct a tree of various related classes, even though some have different properties than others and some

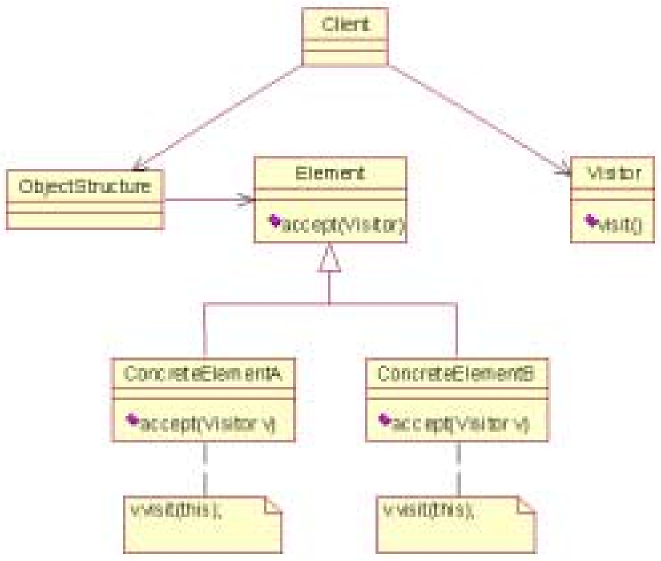

Figure 4: Composite Pattern - Object Model

are leaves that do not have children. However, for very simple cases you can sometimes use a single class that exhibits both parent and leaf behavior.

5.2.7 Related patterns

- *Chain of responsibility:* The Chain of responsibility pattern can be combined with the composite pattern by adding child to parent links so that children can get information from an ancestor without having to know which ancestor the information came from
- *High Cohesion:* The High Cohesion pattern discourages putting specialized methods in general purpose classes, which is something that the composite pattern encourages
- *Visitor:* You can use the Visitor pattern to encapsulate operations in a single class that would otherwise be spread across multiple classes

5.3 Where does the children come from?

As we can see, the composite pattern is exactly what we need. Fortunately we have already a file-system structure that is according to our composite pattern, which means we do not have to change anything here. We got a perfect design, a design that an expert would use as well.

Looking once more on our code, we can see that we have a getChild function that returns our children. But where actually does those children come from? Of course, the user creates them. The user creates new files and new directories. We only have to find a way on how to associate those files and directories to our file-system. We do that simply with an adopt function:

Virtual void adopt(Node* child);

The adopt function adds a new child to a directory. In the same way we define a orphan function, which releases a child from a directory in the same way. This is depicted in figure 5.

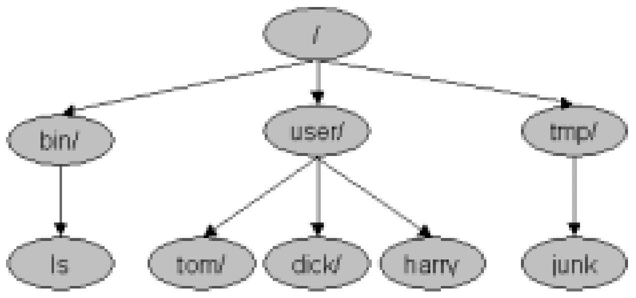

Figure 5: File-System

As we can see, we have implemented the adopt function in our directory class only and not in class node. This has a disadvantage that we are going to show now. Imagine a client that wants to create a new directory. S/he uses a mkdir function (from Unix) to create it. In case that we implement the adopt function only in the directory class as shown in figure 5, we get the following C++ code:

```
void Client::mkdir (Directory* current, const string& path)
{
   string subpath = subpath(path);

   if (subpath.empty())
   {
      current->adopt(new Directory(path));
   }
   else
   {
      string name = head(path);
      Node* child = find(name, current);
      If (child)
      {
         mkdir(child, subpath);
      }
      else
      {
         cerr << name << " does not exist." << endl;
      }
   }
}
```

The intention of the functions head and subpath are here to manipulate strings. head returns the first name of a path and subpath the whole rest. The find function-call searches a directory after a child with a specified name.

As we can see, we create a new directory by specifying the current directory position and the directory that should be created including the full path. With the help of the head and subpath functions we can travel from the current directory to the specified one recursively. Arriving at our destination we create the new path with the adopt function. This code looks simple, nevertheless, it would not compile. The problem is that the recursive call mkdir(child, subpath) passes a child of type node to the function mkdir, which assumes the first parameter to be of type directory.

One solution to this is dynamic down-casting. Adding dynamic casting to our previous code, we get the following modified code:

```
void Client::mkdir (Directory* current, const string& path)
{
   string subpath = subpath(path);

   if (subpath.empty())
   {
      current->adopt(new Directory(path));
   }
   else
   {
      string name = head(path);
      Node* node = find(name, current);

      If (node)
      {
         Directory* child = dynamic_cast<Directory*>(node);
         If (child)
         {
            mkdir(child, subpath);
         }
         else
         {
            cerr << getName() << " is not a directory." << endl;
         }
      }
      else
      {
         cerr << name << " does not exist." << endl;
      }
   }
}
```

Looking at this example, we can say: ”It works”. But where is the snag? The snag is that our class mkdir should try to create a directory and either create it or print out an error message. But this is not the case. The mkdir class has deep knowledge of our file-system structure and checks if the child that it receives is a directory or a file. This is not what we want. Here we break the rules of encapsulation, where one class should not know about the inside structure of another class. What can we do?

As plan A failed, which was to implement the adopt function only into the directory class, we try now plan B, where we add also a virtual adopt and orphan function to our node interface. Concerning our make directory (mkdir) code, we get one main change and that is in line one. We exchange the line to the following one:

```
void Client::mkdir (Node* current, const string& path)
```

This enables us now to specify our current location in form of a node through our interface and no longer just through the directory class. The new UML structure is shown in figure 6.

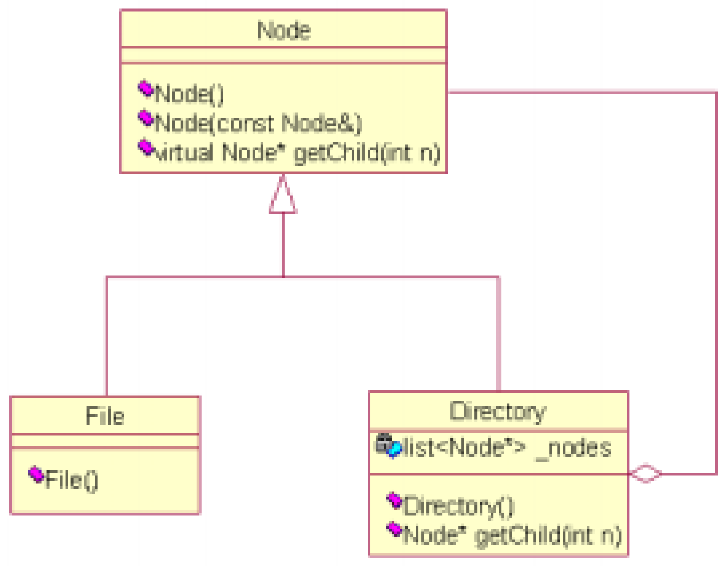

Figure 6: File-System UML Model

5.4 Symbolic Links

Until now we got a complete file-system structure that is designed with the help of the composite pattern. In this part, we want to enhance the system with symbolic links. A symbolic link is in principle only a reference to another node in the file-system. Deleting this link, makes it disappear without effecting any nodes that it referenced to.

A symbolic link has its own access right, which differ from the ones of the referenced node. Otherwise, the link behaves like a usual node. In case the symbolic link refers to a file a user can use this link as a file itself; the user can for example edit or save a file through its symbolic link. On the other hand, if the link refers to a directory, a user can add or delete nodes to the directory in the way that s/he calls the specific operations in the same way for the link as for the directory itself.

Now as we have stated the problem that has to be designed, the question is: Is there a design pattern that describes a solution for exactly this kind of problem? And the answer is yes, the proxy pattern does exactly that. Before we continue with our example, the next section explains the proxy pattern first more deeply.

5.5 Proxy Pattern

5.5.1 Description

Proxy is a very general pattern that occurs in many other patterns, but never by itself in a pure form. The proxy pattern forces method calls to an object to occur indirectly through a proxy object that acts as a gate for the other objects, delegating method calls to that object. Classes for proxy objects are declared in a way that usually eliminates client objects' awareness that they are dealing with proxy.

5.5.2 Forces

- It is not possible for a service-providing object to provide a service at a time or place that is convenient.
- Gaining visibility to an object is nontrivial and you want to hide that complexity
- Access to a service providing object must be controlled without adding complexity to the service providing object or coupling the service to the access control policy.
- The management of a service should be provided in a way that is as transparent as possible to the clients of that service.

5.5.3 Structure

Transparent management of a service providing object can be accomplished by forcing all access to the service providing object through a proxy object. In order for the management to be transparent, the proxy object and the service providing object must either be instances of a common super class or implement a common interface, as shown in figure 7.

5.5.4 Participants

- Proxy
  - administrates a reference, which enables the proxy to access the real subject. The proxy can use the interface of the subject if it is identical with that one from the real subject.
  - controls the access to the real subject and is possibly responsible to create and delete it.
- Subject
  - defines the common interface of real subject and proxy, so that a proxy can be used everywhere, where a real object is expected.

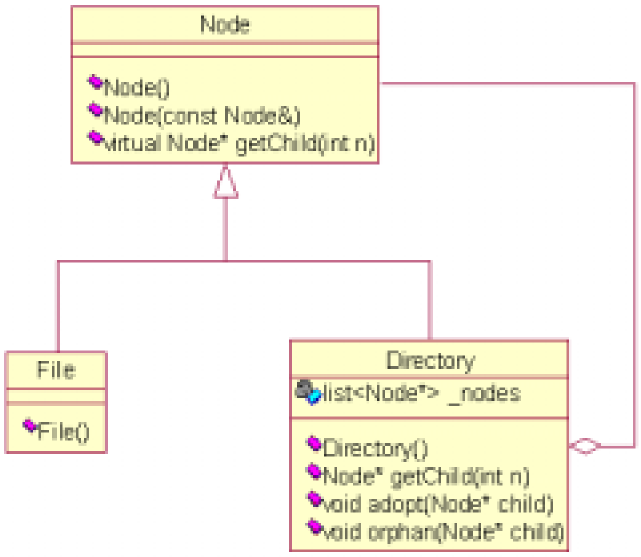

Figure 7: Proxy Pattern

- Real Subject
  - defines the real object, which is being presented by the proxy.

5.5.5 Consequences

The service provided by a service providing object is managed in a manner transparent to that object and its clients. Unless the use of proxies introduces new failure modes, there is normally no need for the code of client classes to reflect the use of proxies.

5.5.6 Implementation

Without any specific management policy, the implementation of the proxy pattern simply involves creating a class that shares a common superclass or interface with a service providing class and delegates operations to instances of the service providing class.

5.5.7 Related patterns

- *Access Proxy:* The access proxy pattern uses a proxy to enforce a security policy on access to a service providing object.
- *Broker:* The proxy pattern is sometimes used with the broker pattern to provide a transparent way of forwarding service requests to a service object selected by the broker/proxy object.
- *Faade:* The faade pattern uses a single object as a front end to a set of interrelated objects.

- *Remote Proxy:* The Remote Proxy pattern uses a proxy to hide the fact that a service object is located on a different machine from the client objects that want to use it.
- *Virtual Proxy:* This pattern uses a proxy to create the illusion that a service providing object exists before it has actually been created. It is useful if the object is expensive to create and its services may not be needed.
- *Decorator:* The Decorator pattern is structurally similar to the proxy pattern in that it forces access to a service providing object to be done indirectly through another object. The difference is a matter of intent. Instead of trying to manage the service, the indirection object in some way enhances the service.

5.6 Composite and Proxy work together

Now as you know more about the proxy pattern, we have to find a way, how to combine the idea of the symbolic link as a proxy pattern with our file-system structure that is presented as a composite pattern. The first thing to do is therefore to find a common structure for the proxy and the composite pattern.

Mapping the proxy pattern to our file-system, we recognize that node remains our common interface. From here we need to inherit the proxy class from our already specified node class. The link class might look as follows:

```
class link : public node
{
   public:
      Link(Node*);

      // other operations

   private:
      Node* _subject;
};
```

The object variable _subject adds a reference to the link class that points to a real subject. Indeed the subject here could either be the file class or the directory class, but in our case it is both. We need a link to directories and to files and therefore the easiest way to conduct this is to point _subject to the class node. Combining the composite and the proxy pattern in their common interface node, we receive a new UML model that is depicted in figure 8.

5.7 Code Enhancement

Our file-system looks pretty nice until now. The whole system consists of two patterns. Next we would like to enhance our system. The system should be able to print out the content of files. The first way we might think to handle this is to add a new functionality to the file class to print out the content of a file. This would mean that we would need to change the file and the node class. Here we should remember a golden rule for software as well as hardware

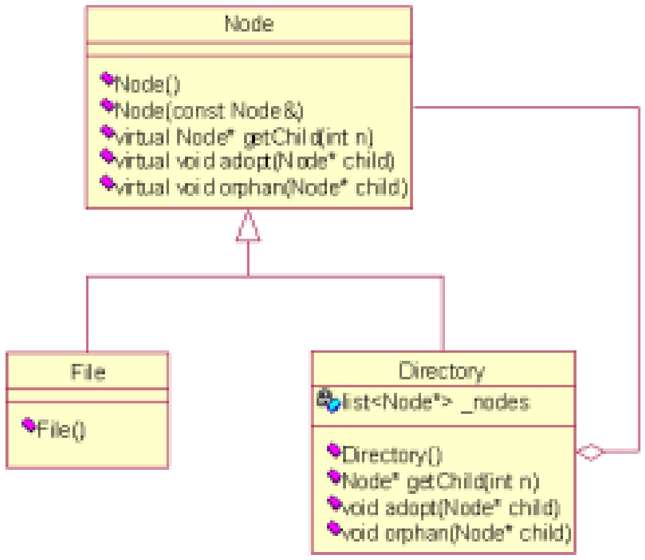

Figure 8: Composite and Proxy Pattern

systems: Never touch a running system". So how do we add the functionality to the file class without changing our beautiful constructed file-system?

One idea could be to have the print-out function outside of our system and implement it in another class that we call the Client. The function on the other hand we call cat. The code to print out the content of a file from outside our system could look as follow:

```
void Client::cat (Node* node)
{
   Link* l;

   If (dynamic_cast<File*>(node))
   {
      node->streamOut(cout);
   }
   else if (dynamic_cast<Directory*>(node))
   {
      cerr << "cat can not be performed on a directory." << endl;
   }
   else if (l=dynamic_cast<Link*>(node))
   {
      cat(l->getSubject());
   }
}
```

This implementation would work but again we face the same problem as before. The question is again: Does an outside class should now so much implementation details about our file-system structure? Should it know to what types to downcast and break our encapsulation? Definitely not! What we need is another solution, maybe another pattern. Studying the design pattern catalogs once more we find a matching pattern for our problem: the visitor pattern. Read more about it in the following part.

5.8 Visitor pattern

5.8.1 Description

One way to implement an operation that involves the objects in a complex structure is to provide logic in each of their classes to support the operation. The Visitor pattern provides an alternate way to implement such operations that avoids complicating the classes of the objects in the structure by putting all of the necessary logic in a separate Visitor class. The Visitor pattern also allows the logic to be varied by using different Visitor classes.

5.8.2 Forces

- There are a variety of operations that need to be performed on an object structure
- The object structure is composed of objects that belong to different classes.
- The types of objects that occur in the object structure do not change often and the ways that they are connected are consistent and predictable.

5.8.3 Structure

Figure 9 shows the structure of the Visitor pattern.

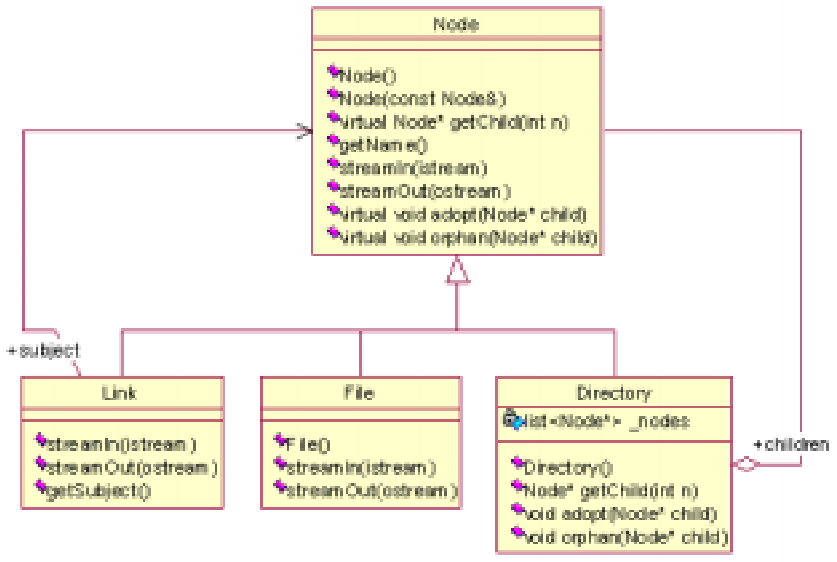

Figure 9: Visitor Pattern

5.8.4 Participants

- Visitor
  - declares a visit operation for every concrete-element class in the object structure. The operations name and its signature give the class a name, which the visitor operation will call. This enables the visitor, to obtain the concrete class of the visited element. The visitor can then under the use of the concrete interface access the element.
- Element
  - defines a accept-operation, which takes a visitor as an argument
- Concrete Element
  - implements the accept-operation, which can take a visitor as an argument.
- Object Structure
  - can enumerate its elements
  - delivers possibly a abstract interface, which allows the visitor to visit its elements
  - can be also a composite

5.8.5 Consequences

The Visitor pattern is useful when you want to encapsulate fetching data from a number of instances of several classes. Design Patterns suggest that the Visitor can provide additional functionality to a class without changing it. However, it's preferable to say that a Visitor can add functionality to a collection of classes and encapsulate the methods that it uses. The Visitor is not magic, however, and cannot obtain private data from classes; it is limited to the data available from public methods. This might force you to provide public methods that you would otherwise not provide.

It is easy to add new operations to a program using Visitors, since the Visitor contains the code instead of each of the individual classes. Further, Visitors can gather related operations into a single class rather than forcing you to change or derive classes to add these operations. This can make the program simpler to write and maintain.

5.8.6 Implementation

You should consider using a visitor pattern when you want to perform an operation on the data contained in a number of objects that have different interfaces. Visitors are also valuable if you must perform a number of unrelated operations on these classes. Visitors are a useful way to add function to class libraries or frameworks for which you either do not have the source or cannot change the source for other technical (or political) reasons. In these latter cases, you simply subclass the classes of the framework and add the accept method to each subclass. Visitors are a good choice, however, only when you do not expect many new classes to be added to your program.

5.8.7 Related patterns

- *Iterator:* The Iterator pattern is an alternative to the Visitor pattern when the object structure to be navigated has a linear structure.
- *Little Language:* In the Little Language pattern, you can use the Visitor Pattern to implement the interpreter part of the pattern.
- *Composite:* The Visitor pattern is often used with object structures that are organized according to the Composite pattern.

5.9 The Visitor Pattern joins our file-system

As we know now more about the visitor pattern, we know also that what we need to do first is to extend all our classes, including the node interface with an accept function as follows:

```
virtual void accept(Visitor&)=0; // Node

void File::accept (Visitor& v)         { v.visit(this); }
void Directory::accept (Visitor& v)     { v.visit(this); }
void Link::accept (Visitor& v)         { v.visit(this); }
```

With the help of our accept functions we can now write a visitor class that can visit a specified node in order to perform some operation. In our case we want to print out the content of a file if the node is of the type file, else we print out an error message.

```
class Visitor
{
   public:
      Visitor();

      void visit(File*);
      void visit(Directory*);
      void visit(Link*);
}

void Visitor::visit (File* f)
{
   f->streamOut(cout);
}

void Visitor::visit (Directory* d)
{
   cerr << "cat can not be performed on a directory" << endl;
}

void Visitor::visit (Link* l)
{
   l->getSubject()->accept(this);
}
```

Finally, we can specify the code that creates us a visitor object and visits the accept function of our file-system:

```
Visitor cat;
node->accept(cat);
```

In case the node is of type file, we call the accept function in the file class. This function on the other hand has only one implementation:

```
v.visit(this);
```

This code calls our visitor object cat that we have passed to the file object. The function visit is called with the file object as an parameter. After a type check of the overloaded function Visitor::visit, the right function for handling files is found and the content of the file is printed.

In that way we have added new functionality to one or more classes and we have specified a new class that handles all the internals of this new functionality. In this case the new class actually knows again some internals about our file-system structure, like it knows about the file, directory and link classes, but in this case it is ok, since the visitor class is a part of our file-system, while the Client class is not. We can see that we actually have added another pattern to our file-system. The complete system, consisting of the composite, the proxy and the visitor pattern is depicted in figure 10.

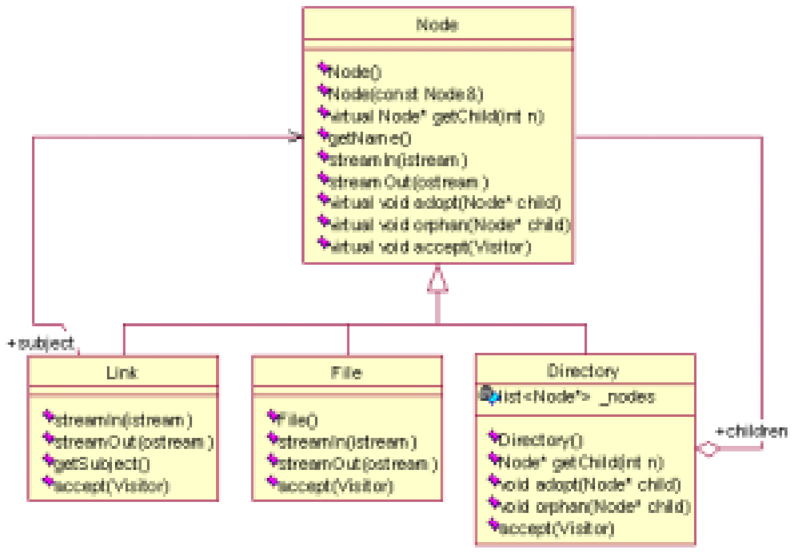

Figure 10: 3 Patterns combined

References

[1] Entwurfsmuster, Elemente wiederverwendbarer objectorientierter Software, Erich Gamma, Richard Helm, Raplh Johnsson, John Vlissides, Addison-Wesley, 5 Auflage, 2001, ISBN 3-8273-1862-9

[2] Design Patterns, Erich Gamma, Richard Helm, Raplh Johnsson, John Vlissides, 1995, ISBN 0-201-63361-2

[3] Java Design Patterns, A Tutorial, James W. Cooper, 3rd version, July 2000, ISBN 0-201-48539-7

[4] Patterns in Java, A Catalog of Reusable Design, Patterns Illustrated with UML, Mark Grand, 1998, ISBN 0-471-25839-3

[5] Entwurfsmuster anwenden, John Vlissides, Addison Wesley, 1999, ISBN 3-8273-1544-1

[6] Pattern hatching, Design Patterns applied, Addison Wesley, 1998, ISBN 0-201-43293-3