Patterns for Encapsulating Class Trees

by
Dirk Riehle
Union Bank of Switzerland

Introduction

Good object-oriented design firmly relies on abstract classes. They define the interface to work with subclasses that implement them. If clients directly name these subclasses, they become dependent on them. This complicates both system configuration and evolution. The patterns of Late Creation and Class Retrieval presented in this paper overcome these problems by encapsulating class trees behind their root classes. Clients use class specifications to retrieve classes and create objects. Classes can be removed and plugged into the class tree more easily. Thus, encapsulating class trees eases system evolution and configuration of system variants.

The patterns of this paper are presented using a new variation of the pattern form. We start by outlining the overall background on which the patterns emerge. This background explains what the patterns are relevant for and what they achieve. It is the overall pattern context and the recursive closure of all subsequent pattern contexts.

After introducing the background, we present each pattern in a subsection of its own. A subsection mainly consists of a pattern/context pair that separates the actual pattern from the embedding context. This is based on our understanding of a

In: *Pattern Languages of Programm Design 2.* Chapter 6.
Edited by John M. Vlissides, James O. Coplien and Norman L. Kerth.
Reading, Massachusetts: Addison-Wesley, 1996.

pattern as a *form* emerging in specific contexts. The form is finite and can be described precisely, while the context is infinite and can only be partially described (that is, we extract what we think is relevant to understand the forces driving the pattern).

Section 2 introduces the background of the presented patterns. Section 3 and 4 present the patterns needed for encapsulating class trees. Section 5 compares the patterns with other patterns, most notably Factory Method and Abstract Factory. Section 6 goes back to discuss the pattern form and what we have gained from using it. Section 7 summarizes the paper and presents some further conclusions.

Background of the Patterns

Object-oriented design firmly relies on abstract classes. They represent the key design decisions that structure a system in the large. An abstract class represents the interface to a whole class tree. This interface is often sufficient for clients to work with objects of the class tree. Only for selecting classes of the class tree and creating objects of them, concrete subclasses of the abstract class have to be named. If this is done by clients of the abstract class, they become dependent on the class tree's internal structure. Changes of class names and class tree structure force clients to be changed.

Therefore, the class tree behind an abstract class should be hidden from all clients of that class. If a client's knowledge is restricted to the abstract class only, all dependencies on subclasses are cut. This has several advantages. First, clients can focus on the relevant abstraction, that is the abstract class. They don't have to bother with less important details like names of subclasses. Second, changes of the class tree have only local consequences which makes system evolution easier. Third, plugging in and removing subclasses doesn't affect clients. Thus, system variants can be configured in an easy way.

The general idea for encapsulating class trees is to let clients refer to the abstract superclasses only. They retrieve classes and create objects by supplying class specifications. A specification describes the classes of interest to the client and makes up for the information loss of class names. A common meta facility maps these specifications onto classes. This task is conveniently carried out by the abstract superclasses of a class tree themselves. These superclasses visible to clients are called *interface classes* of the class tree. While working with encapsulated class trees, two different but related patterns emerged: *Class Retrieval* and *Late Creation*.

Class Retrieval is the process of retrieving a set of classes from an encapsulated class tree which all adhere to a given specification. This set can be used for further decisions on how to proceed. For example, retrieving all classes in an encapsulated *Command* class tree [GHJV95] can be used for building a menu item list of possible commands.

Late Creation is the process of creating a single instance of a class which matches a given specification. Thereby, instances of hidden subclasses can be created using interface classes only. The most common example is object activation. A class id received from a stream is mapped to a class which is used to create the new object. Late Creation lets clients not only perform object activation based on class id's but create new objects based on all kinds of specifications. Such specification might depend on existing objects and can therefore be used to create a set of objects in concert, for example from a behavioral pattern.

Several ways of specifying classes can be thought of, most prominently interface definition languages and object request brokers. Here, however, additional forces are introduced: Class Retrieval and Late Creation have to be carried out fast and in the order of time used by *Factory Methods* [GHJV95]. Achieving this, they can be incorporated into basic framework design and may be used as pervasively as factory methods.

The patterns *Class Specification*, *Class Semantics* and *Class Clause* support Class Retrieval and Late Creation and achieve the desired order of speed. Class Specification is used to express requirements for classes to be retrieved or new objects to be created. Class Semantics is used to express properties of a specific class so that it can easily be matched with a specification. A Class Clause expresses an atomic property of a class as a first class object. Figure 1 shows an overview of the patterns and their dependencies.

The patterns have been implemented in two different frameworks, one in Smalltalk [RS95] and one in C++ [RZ95]. The example designs from these frameworks used to discuss the patterns are based on a simple metalevel architecture which assumes some kind of representation of classes as objects so that classes can be passed around. This is available in most major C++ application frameworks [WG94, GOP90, CIRM93] as well as in CLOS [Ste90] and Smalltalk [GR83].

Patterns for Encapsulating Class Trees

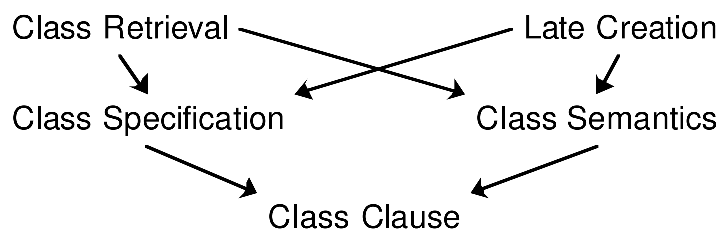

Figure 1: Overview of the patterns. An arrow expresses that the source pattern relies on the target pattern of the arrow. Therefore, it provides a possible context of its use.

Framework Example

As an example, consider a system designed according to the Tools and Materials Metaphor [RZ95]. Users use *tools* to work on *materials*. Tools are means of work while materials are the intermediate or final outcome of work. A software desktop presents both tools and materials as icons to users. They start tools by clicking on the corresponding icon.

We will deal with two tools from a time planning system: a calendar tool that works with an appointment book and a planner tool that works with a time table. The appointment book as well as the time table are materials. The calendar is used to keep track of single appointments while the planner is used to manage weekly dates like meetings and seminars.

Figure 2 and 3 show a simple software design. The classes `Calendar` and `Planner` are subclasses of the abstract superclass `Tool`. The classes `AppBook` and `TimeTable` are subclasses of the abstract superclass `Material`. In a running system, a single instance of class `Desktop` creates, manages and deletes all tools and materials. We simplify the example by assuming that all classes are available in a single executable. To focus on the essentials, we ignore issues of dynamically linked libraries and interprocess communication.

What does it mean to encapsulate class trees for this example? Systems designed according to the Tools and Materials Metaphor usually consist of a large and changing number of tools and materials which are supplied with a specific system variant. Therefore, the class `Desktop` should not know about specific tools but should use the abstract superclass `Tool` only. Thus, the class trees behind

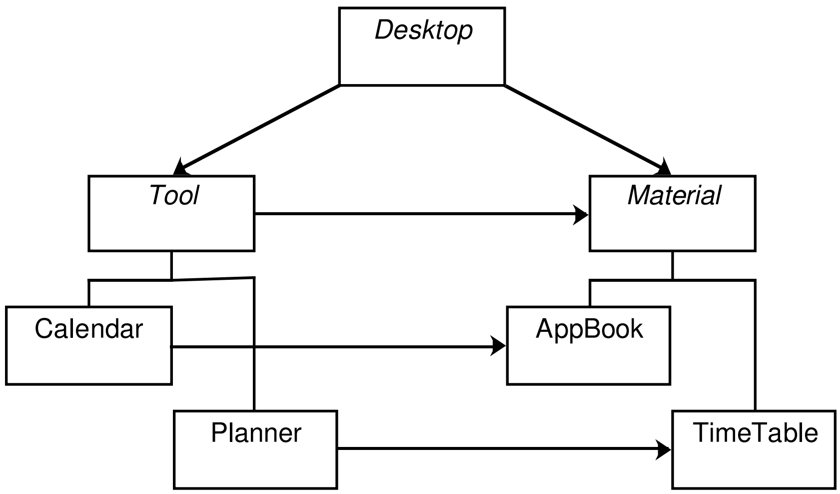

Figure 2: *Software design of the example. Rectangles represent classes, arrows use and lines inheritance relationships.*

`Tool` as well as `Material` should be encapsulated for `Desktop`. Doing so, several tasks have to be reconsidered:

- Each tool in the system should be represented as an icon on the desktop. How can the desktop ensure that the represented tool is actually available in the current system variant?
- Assume that the user selected a tool icon or typed in a tool name. How can the desktop determine the corresponding tool class and create an actual tool instance?
- Assume that the user double clicked on a material to start it up. Usually a large number of tools can work on a material in more or less specific ways (see pattern Tool and Material Coupling in [RZ95]). How can the desktop find out the best fitting tool class to work on the material?

These questions will be successfully answered in the example sections of the patterns of the next two sections.

Class Tree Encapsulation

Clients of an encapsulated class tree may not directly refer to the internal classes of the tree. At runtime, however, they have to determine the available classes so that they can rely on them. Furthermore, they have to create objects from these hidden classes to actually make use of them. *Class Retrieval* shows, how clients retrieve sets of classes that all adhere to a given specification. *Late Creation* shows, how clients create objects from hidden classes also using specifications.

Class Retrieval

Clients of an encapsulated class tree need to retrieve classes from that tree. They delegate the task to the interface class of the tree and provide it with a class specification. They receive a set of classes all adhering to the specification which they use according to their purposes.

Context

Several tasks require knowledge about the available classes in a class tree. For example, clients need to know the available classes in a *Command* [GHJV95] class tree so that they can build a list of menu items for them. Or clients want to know all classes in a *Strategy* [GHJV95] class tree that perform with certain timing characteristics.

Clients of an encapsulated class tree may not know these classes by name. They only know the interface class of the class tree. However, they know the properties of those classes which they need for their task. Therefore:

Pattern

Clients of an encapsulated class tree create a specification for those classes they are interested in. The specification is based on the properties that are relevant to the client. The client requests all classes from the class tree that adhere to the specification. This task is most conveniently delegated to the interface class of the class tree. The client receives a set of all classes in the class tree that match the specification.

All classes in the set are subclasses of the interface class and therefore guaranteed to support its interface. The client can use this set according to its purposes, for example to present each class as a possible choice in a graphical user interface. Specifications for Class Retrieval are usually ambiguous specifications which denote a whole range of possible classes.

Example

The desktop class has been written to work generically with tools. It doesn't know about specific tool classes. The tool classes vary with each system variant. Thus, during each system startup, the desktop has to determine which tool

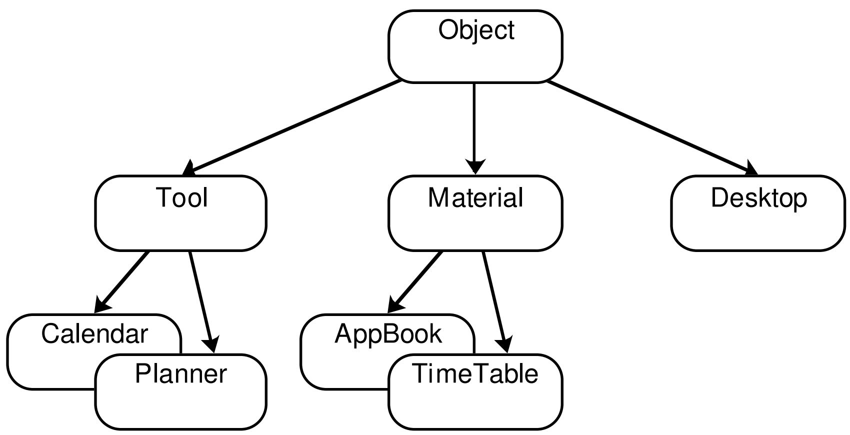

Figure 3: *The class (object) tree of the example. Each class holds a list of its subclasses. Objects are shown as rounded rectangles.*

classes are available. Icons should only be created for tool classes that are actually available.

Therefore, the desktop creates a specification for concrete tool classes. The specification consists of a single flag which indicates that the classes have to be instantiable. It calls the `RetrieveClasses` operation of class `Tool`. A set of classes that fit the specification is returned (see the following code example). Among others, it contains the classes `Planner` and `Calendar`. The desktop now can safely create icons for these tool classes.

```
void Desktop::GetConcreteToolClasses( Set< Class* >* cset )
{
      // create simple specification for concrete classes
   IsAbstractClause spec( false );

      // request classes from interface class
   Tool::ClassObject()->RetrieveClasses( &spec, cset );
}
```

Design

At least two possible implementations come to mind. The interface class can build the set by traversing the class tree top down. While traversing, it matches each class with the specification. If the specification fits, the corresponding class is put into the set which will be returned to the client. This is a slow implementation but it introduces no additional memory overhead. The patterns of chapter 3 discuss how object-oriented specifications can be built easily.

If speed is more important, the specification can be used to compute the index for a table lookup. Each specification usually denotes a set of classes, which means that the index for the table lookup identifies the set of equivalent classes for a certain specification. These tables can be built in advance or on demand.

The execution time is now constant but requires some memory overhead for the tables. Our implementations use class tree traversal for *Class Retrieval* and table lookup for *Late Creation* (next pattern). It doesn't compute tables for all kinds of specifications (which would be impossible), but only for specifications built from a single clause (see pattern Class Clause).

Impact

The code for retrieving classes can fully be written on the framework level. It usually consists of two lines, one creating a specification, one calling the retrieve classes operation. Retrieving classes requires additional functionality which depends on the chosen implementation strategy. This is solved by the framework and shouldn't bother users. The only task left to users of the framework is to specify the semantics of the classes which they introduce. This effort is in the order of writing an access operation for each property specified (see the patterns of the next section).

Late Creation

Clients of an encapsulated class tree need to create objects of classes hidden in that tree. Again, clients create a specification for the objects to be created and delegate the task to the interface class. They either receive an instance of the class matching the specification or null.

Context

An encapsulated class tree is of no use unless objects of its internal classes can be created. Again, clients know only the interface class. In addition, they now have to know a property that unambiguously identifies the class they are interested in. For example, a class id received from an input stream has to be mapped to a class to activate an object. Therefore:

Pattern

The client of a class creates a specification that unambiguously identifies a single class. It then requests a new instance of a class that fits the specification. Again, this task is conveniently delegated to the interface class. A returned object is guaranteed to both adhere to the interface class and the specification. If no matching class is found, no instance can be created and null is returned.

The most widely known example of Late Creation is object activation. Class names or class identifiers received from a stream are mapped to a class which then is used to create an object. Late Creation is the generalization of several special purpose solutions existing today [Gro93, GOP90]. Using table lookup it can be carried out in constant time.

The name “Late Creation” resembles the notion of late binding. As with late binding the class which eventually is referred to is determined at runtime.

Example

Tools should have unambiguous names like “Calendar” or “Planner.” Assume that the system can be started using aliases. The chosen alias indicates the first tool to be launched automatically. Thus, the desktop creates a specification for a tool class by using the alias string. Any unambiguous specification (class id, class name, tool name) suffices.

The desktop calls the `CreateLate` operation of the abstract class `Tool`. `Tool` uses the specification to retrieve the corresponding subclass. `Tool` asks this class to create an instance of it which is returned to the client. If no matching class is found, null is returned.

Design

Again, it is possible to both traverse the class tree or to use table lookup to find a class. In this case, given an unambiguous specification, a table entry denotes a single class. Thus, the table can be built more easily. Developers should take care that during evolution the system stays free of ambiguities. We solve the issues by carefully designing and inventing new classes, however, tool support is certainly desirable.

If clients are allowed to pass in ambiguous specifications, the class of the new instance can’t be decided without further help. Thus, some means of handling ambiguities has to be introduced. Clients might supply additional hints like “choose the most specialized” or “most general implementation.” Or they will just receive an instance of the first class that is retrieved.

Impact

Class Retrieval and Late Creation have similar impacts. Again, the execution speed depends on the chosen implementation strategy and can be done in constant time in the order of a factory method.

For certain classes, hiding the creation process from the client might turn out to be a problem. Inside the class tree, objects can only be created using a standard initialization procedure. If the new object needs additional parameters, it must receive them from the client through an extra operation after creation. However, the protocol of this extra initialization operation has to be available in the interface class of the class tree and no assumption about class tree internal protocols should be made.

In very problematic cases, the specification can be enriched with the initialization parameters, so that a special creation procedure or constructor can receive them during the early object building process.

Specification Support

This section discusses class specifications and class semantics. Specifications for a class as well as class semantics are built from clauses. A clause is an atomic predicate about a class property. The semantics of a class are represented by a set of clauses. A class specification is an expression from propositional calculus with clauses as its atomic constituents. Class specifications can be matched against class semantics in a straightforward fashion.

Class Clause

A class clause makes an atomic statement about a class property. It resolves to either true or false. It is represented by an object and can be compared to other clauses easily. Thus, it provides a basis for class specifications and first class representations of class semantics.

Context

Class Retrieval and Late Creation use specifications to describe classes of interest. Clients of an encapsulated class tree build specifications for one or more classes. Thus they need some basic means to express class properties. Furthermore, designers of a class wish to make its properties explicit as well. Thus, they also need a basic means to express class properties. Finally, we have narrowed the context to the programmatic statement that specifications should be easy and need no further language or tool support. Therefore:

Pattern

Class properties are expressed as instances of clause classes. Each clause class picks on a specific aspect of a class's semantics and represents it as an object. Clients use clause class instances to build specifications. A single clause is a special case of a general class specification and can be used as such very well. Furthermore, classes use clauses to express their basic properties.

Several kinds of clause classes exist. Some express general properties, for example whether a class is abstract or concrete. Others express performance or memory consumption properties of instances of a class. Some clauses denote a class unambiguously, for example clauses built from class id's, class names or other identifiers. Clause classes are introduced as needed.

The class tree of figure 4 shows some of the most convenient clause classes. All clause classes are subclasses of `Clause`. Those classes that offer an unambiguous id are subclasses of `IdClause` that in turn is a subclass of `Clause`. Further clause classes are `IsAbstractClause` (a flag indicating abstract or concrete classes), `NameClause` (a string indicating a class) and several kinds of id clauses.

Next to static properties of a class, clauses can express dependencies between several classes. A `MaterialIdClause` instance holds the id of a material class. It can be used as a simple specification for tool classes that can work with the indicated material. Such clauses are called *dependency clauses*.

When created, a clause instance receives its parameters from its creator. If the client is using the clause for a specification (see next pattern, Class Specification), the client has to explicitly provide the class properties expressed through the clause. If the clause is instantiated to be part of a class's semantics, it might derive the properties from the class itself (see next but one pattern, Class Semantics).

Example

The specification for concrete tool and material classes is based on the clause class `IsAbstractClause`. It consists of a flag that indicates whether a class is abstract or not. The specification for object activation consists of a single `ClassIdClause` instance that holds the id for the class to be instantiated.

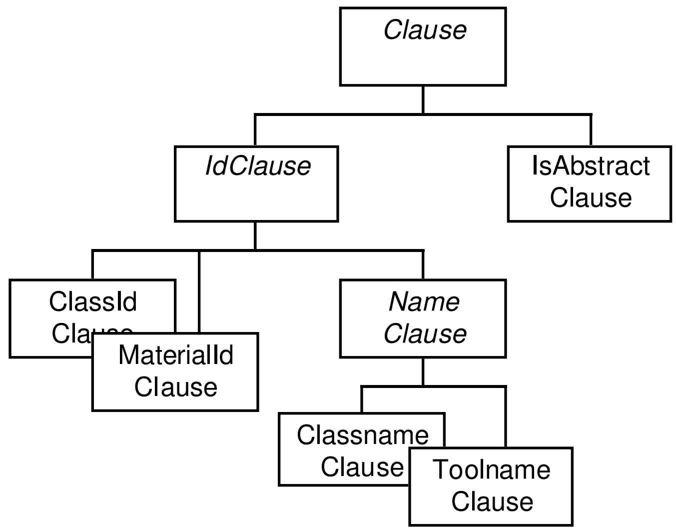

Figure 4: *Hierarchy of clause classes. Most clause classes are subclasses of* `IdClause` *and thus are capable of denoting a single class.*

As another example, assume that the user activated a material instance of class `AppBook`. Now a tool to work on the given material has to be instantiated. The desktop creates a `MaterialIdClause` instance which it provides with the class id retrieved from the appointment book. It calls `CreateLate` of class `Tool`. The `MaterialIdClause` instance is matched by the tool class `Calendar` as the most specialized class being able to work on the indicated material. From the desktop's point of view it has created a tool instance depending on a given material. Only the abstract superclasses `Tool` and `Material` were used. Thus, the class trees are encapsulated.

As yet another example consider a client that holds a `List` instance and wants to create an iterator for it. It calls the `CreateLate` operation of class `Iterator` and passes in a `ContainerIdClause`. This clause instance holds the class id of the container class, that is the list class id. Each iterator class knows the container it has been designed for and can therefore decide whether it matches the specification. Using clauses this way avoids the usual factory methods of the container to create iterators and cursors of all flavors.

Design

It has been left to prove that the execution time for class lookup can be in the order of time consumed by factory methods. Given a specification consisting of a single clause and a prebuilt table for looking up classes based on this clause class, the following steps have to be taken: Create a clause instance, lookup the right table based on the clause class, lookup the class based on the clause, return it or call new on it. The following code example shows a simple C++ operation for `CreateLate`.

```
Object* Class::CreateLateByClause( IdClause* clause )
{
      // get clause instance table for the clause's class
   Table* cltab = ClauseTab->Lookup( clause->ClassObject() );

      // lookup class for given clause
   Class* classObject = cltab->Lookup( clause->Id() );

      // return new object of retrieved class
   if ( classObject != null ) return classObject->New();
   else return null;
}
```

Class Specification

A class specification is a formula from propositional calculus with clauses as its basic constituents. Clients build a specification by creating clauses and arranging them as a formula.

Context

Clients use specifications to retrieve classes from a class tree. They use clauses to express basic properties of these classes. However, single properties are often not sufficient to give a precise specification for a class. Therefore:

Pattern

A class specification is a formula from propositional calculus. Its atomic predicates are clauses that express simple class properties. A formula lets clients combine these clauses in a way sufficient for all tasks we encountered so far. The formula is built from clauses using the standard operators of propositional calculus (and, or, not).

An important case of a class specification is a *dependency specification* built from dependency clauses: Consider an abstract design consisting of more than a single class. These classes are often subclassed in concert to take advantage of each other. Therefore only specific subclasses work together. Only specific iterators match specific containers. A new object that is to participate in the design has to match the already existing objects. A specification for this object is a dependency specification that consists of a conjunction of id clauses, one for each existing object. Each class can check whether it can work with the classes indicated in the clauses. If so, it will match the specification.

Example

Full specifications are often a combination of clauses. For example, the desktop might wish to retrieve all concrete tool classes that work with a certain material class. The resulting specification is a conjunction of an `IsAbstractClause` (indicating that the class has to be concrete) and a `MaterialIdClause` instance (indicating the material the tool has to work with). Asking class `Tool` for all concrete subclasses that can deal with a `TimeTable` material will result in quite a large set of tool classes. The set will include some general lister and browser tools as well as the more specialized planner tool.

Design

A specification is an instance of `ClassSpec` which arranges the clause instances it receives as a formula from propositional calculus. Very often, however, a class specification consists of only a single clause. Therefore, interface classes should provide both `RetrieveClasses` and `CreateLate` operations for `ClassSpec` and `Clause` instances respectively.

Impact

The execution time required to evaluate a complex formula is still fast compared to interpretative approaches, however, it can't be compared with factory methods anymore. If this turns out to be problematic, a new clause class can be created that expresses the formula as a single clause. The logic has to be realized by program code and thus is as efficient as possible.

Class Semantics

For each class a set of clause instances is provided. Each clause makes a statement about the class it has been instantiated for. The set of clauses is said to represent a class's semantics. A class can be matched against a specification, which is realized by comparing clauses and evaluating the formula.

Context

Class specifications are built from clauses. Classes have to be matched against such specifications as described in Late Creation and Class Retrieval. The matching process should be simple and fast. Therefore:

Pattern

For each class in the system a set of clause instances is provided. Each clause instance makes a specific statement about a property of the class it is maintained for. Thus each clause instance stands for a property of the class it has been designed and instantiated for. It is sensible (though not necessary) to have a class maintain the set of clause instances describing its properties.

At least two rules based on the substitutability principle and deterministic semantics [Lis88, LW93] apply:

- If a class holds an instance of a specific clause class, than each subclass must also hold an instance of that clause class.
- If a clause instance makes a definitive statement about a class property, than this property cannot be changed in subclasses.

Clause classes should be designed with the substitutability principle in mind.

A specification is matched against a class by successive matching of clauses and evaluating the formula. If a clause from a specification isn't found in the class's set of clauses, the clause evaluates to false. However, asking a class tree about clauses it doesn't know usually indicates a design flaw.

Example

The system's root class, `Object`, holds instances of `IsAbstractClause`, `ClassIdClause` and `ClassnameClause`. Each subclass also holds instances of these classes. Class `Tool` holds an additional `ToolnameClause` and a `MaterialIdClause` instance.

Design

Some managing facility has to be introduced which sets up the structure described above. In the C++ framework we use the class `Metaclass` as a managing facility of its instances, which are the class (objects), and in the Smalltalk framework we use an additional `ClassManager`. In C++ it was easy to introduce some kind of metaclass (though not full-blown, of course), in Smalltalk we didn't want to interfere with the standard metalevel architecture.

This managing facility associates a set of clause classes with each class and creates instances of them. Each clause class is associated with a root class it can make a statement about. Its instances are spread by the managing facility to the whole class tree of the root class.

Impact

A concrete design has to clarify where clause instances get their data for describing class properties from. They may be retrieved from external databases or hold in some meta information provided by the class. The simplest approach (and the one we have used so far) is to make the class itself provide the data that clause instances rely on. This forces clients to write simple access methods for each class that return the data needed by the clauses. Each tool class, for example, has a `GetMaterialClass` operation that returns the material class the tool has originally been written for. This simple approach requires some discipline but has worked well for us, mainly because not every new class requires a full set of access methods but can rely on those it inherited. They provide its "default semantics."

At first glance one might imagine that with hundreds or possibly thousands of classes memory problems lie straight ahead. However, only a very small number of clause instances are hold by each class (only `IsAbstractClause`, `ClassIdClause` and `ClassnameClause`). `MaterialIdClause` instances, for example, are maintained only for tool classes, which is just a fraction of the overall number of classes in our systems.

Moreover, clause instances usually aren't fat but consist of 1 up to 4 or 8 bytes. Memory consumption is fixed and can be calculated. We haven't run into trouble with our desktop applications. In case of trouble, optimization strategies can easily be thought of, for example by using *Flyweights* [GHJV95] to minimize the number of clauses.

Relation to Other Patterns

The patterns presented here are an alternative to two of the most important patterns known today: *Factory Method* and *Abstract Factory* [GHJV95].

A factory method is an operation which creates an object that fits the current class. It is destined to be redefined in subclasses where it creates an object that fits the subclass best. Using *Late Creation* and *Class Retrieval*, the factory method still exists, however, it needn't be redefined. The creation process is carried out on the abstract level using a specification for the new object. No redefinition of the operation is needed which possibly avoids introducing a new subclass. Again the desktop might serve as an example. In most designs, it might have been subclassed to introduce the available tool and material classes by name. This isn't necessary as has been shown.

An abstract factory encapsulates the creation process of instances of a family of classes. Some initial specification is used to choose among variants for different systems. The standard example is a window system abstract factory. Based on the runtime environment an abstract factory for either Motif, Macintosh, OS/2 or any other window system is chosen. The factory offers operations to create new windows, scrollbars, text fields, menus and so on. This can easily be reinterpreted in terms of Late Creation. A client that whishes to create a new window uses a window system clause. It consists of a flag that indicates the current window system. The client directly asks the abstract class `Window` to create an instance late. It will receive an instance of a subclass which implements the window for the system indicated in the clause.

In [GHJV95] further creational patterns are listed (Prototype and Builder). They cannot be directly replaced by Late Creation and Class Retrieval. However, they have some similarities. Prototype has similarities with classes as objects and its usage resembles Class Retrieval. Builder does things that can be done using Late Creation as well.

I believe that Late Creation, like Factory Method, is a fundamental creational pattern which can be used to implement the other patterns and works stand-alone as well.

Pattern Form

In this paper we have used a new presentation form based on our experience with patterns.

Each pattern is understood as *form* within a specific non-arbitrary context [Rie95]. This form constitutes the actual pattern and is finite. Therefore we can identify and describe all relevant parts of the pattern. The pattern's context, however, is not finite. We can approach it only pragmatically by describing what we perceive to be the relevant forces giving shape to the actual pattern, that is the form that emerges within that context. Since the understanding of a pattern's context is crucial to understanding the pattern, and since pattern and context have to fit each other, we always describe them together. The result is a pattern/context pair which is supplied for pragmatic reasons with additional sections on examples and design or implementation issues.

Contexts overlap and individual patterns often serve as a part within a whole that is more than just the sum of the single patterns. Therefore we start describing a set of patterns by introducing the background of the patterns. We describe the overall rationale (class tree encapsulation in the case of this paper) and thus provide an embedding and a higher-level understanding of the patterns to follow. This background serves as the closure of the otherwise resulting infinite recursion of embedding pattern/context pairs.

As discussions at PLoP showed, the entry point to a set of related patterns is still considered to be problematic. A “first pattern approach” doesn’t seem to work well since it sets the focus of the following patterns too narrowly on the support of the initial pattern. The notion of background as presented above might be a better alternative.

We have experimented with different pattern forms and think that the presentation form of a pattern depends on its intended use. The problem/context/solution form, for example, seems to be aimed at developing solutions for problems in design. It doesn’t seem to work well when perceiving and identifying patterns in existing structures is more important, for example in legacy systems. A pattern/context pair is more general, since it doesn’t induce specific ways of using the pattern.

Summary and Conclusions

We have presented patterns that let developers encapsulate class trees. A possible design can be based on a simple metalevel architecture which lets developers encapsulate class trees without too much overhead.

Coplien’s generic exemplar idiom [Cop92] is a variant of Late Creation. Lortz and Shin report on concepts for Class Hiding as well [LS94]. Berczuk presents similar patterns [Ber96]. ET++ [WG94] uses a special purpose variant of Late Creation to perform object activation. Several relationships of the patterns to interface definition languages and object request brokers exist. Thus, variants of the patterns presented here have been developed independently around the world. This gives the presented concepts real pattern status.

The patterns work stand-alone, however, tool support for system variant configuration should be provided. Such a tool generates the makefiles that are used to build a specific system variant. The tool should provide support for specifying dependencies between classes, so that no class participating in a design is forgotten. This ensures the absence of runtime failures due to missing classes.

The pattern form has been reduced to a pattern/context pair thereby imposing less structure on the pattern description. However, this is still a time of experimentation and future revisions of our understanding of the pattern form might lead to different and enhanced results.

Acknowledgments

I wish to thank Steve Berczuk, Walter Bischofberger, Brad Edelman and Kai-Uwe Mätzel for reviewing and/or discussing the paper with me. I'd also like to thank the reading group at UBILAB which discussed the paper in a writer's workshop setting. Finally the writer's workshop at PLoP '95 pointed out unclear issues and helped me to improve the paper further.

Bibliography

Ber96

Steve Berczuk. "Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams." This Volume.

CIRM93

Roy H. Campbell, Nayeem Islam, David Raila and Peter Madany. "Designing and Implementing Choices: An Object-Oriented System in C++." *Communications of the ACM 36, 9* (September 1993): 117- 126.

Cop92

James O. Coplien. *Advanced C++: Programming Styles and Idioms.* Reading, Massachusetts: Addison-Wesley, 1992.

GHJV95

Erich Gamma, Richard Helm, Ralph E. Johnson and John Vlissides. *Design Patterns: Elements of Reusable Design.* Reading, Massachusetts: Addison-Wesley, 1995.

GOP90

Keith E. Gorlen, Sanford M. Orlow and Perry S. Plexiko. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons Ltd., 1990.

GR83

Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Reading, Massachusetts: Addison-Wesley, 1983.

Gro93

Mark Grossman. "Object I/O and Runtime Type Information via Automatic Code Generation in C++." *Journal of Object-Oriented Programming 6, 4* (July/August 1993): 34-42.

Lis88

Barbara Liskov. "Data Abstraction and Hierarchy." OOPSLA '87 Addendum, *ACM SIGPLAN Notices 23, 5* (Mai 1988): 17-34.

LS94

Victor B. Lortz and Kang G. Shin. "Combining Contracts and Exemplar-Based Programming for Class Hiding and Customization." OOPSLA '94, *ACM SIGPLAN Notices 29, 10* (October 1994): 453-467.

LW93

Barbara Liskov and Jeanette Wing. "A New Definition of the Subtype Relation." ECOOP '93, *Conference Proceedings.* Berlin, Heidelberg: Springer-Verlag, 1993. 118-141.

Rie95

Dirk Riehle. *Patterns – Exemplified through the Tools and Materials Metaphor.* Masters Thesis, in German. UBILAB Technical Report 95.6.1. Zürich, Switzerland: Union Bank of Switzerland, 1995.

RS95

Dirk Riehle and Martin Schnyder. *Design and Implementation of a Smalltalk Framework for the Tools and Materials Metaphor.* UBILAB Technical Report 95.7.1. Zürich, Switzerland: Union Bank of Switzerland, 1995.

RZ95

Dirk Riehle and Heinz Züllighoven. "A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor." *Pattern Languages of Program Design.* Edited by James O. Coplien and Douglas C. Schmidt. Reading, Massachusetts: Addison-Wesley, 1995. 9-42.

Ste90

Guy L. Steele. *Common Lisp. The Language.* 2nd Edition. Digital Press, 1990.

WG94

André Weinand and Erich Gamma. "ET++ a Portable, Homogenous Class Library and Application Framework." *Computer Science Research at UBILAB.* Edited by Walter R. Bischofberger and Hans-Peter Frei. Konstanz, Germany: Universitätsverlag Konstanz, 1994. 66-92.