How and Why to Encapsulate Class Trees[1]

Dirk Riehle[2]

riehle@informatik.uni-hamburg.de
Software Engineering Group, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany

Abstract

A good reusable framework, pattern or module interface usually is represented by abstract classes. They form an abstract design and leave the implementation to concrete subclasses. The abstract design is instantiated by naming these subclasses. Unfortunately, this exposes implementation details like class names and class tree structures. The paper gives a rationale and a general metaobject protocol design that encapsulates whole class trees. Clients of an abstract design retrieve classes and create objects based on class semantics specifications. Using abstract classes as the only interface enhances information hiding and makes it easier both to evolve a system and to configure system variants.

1 Introduction

A good framework consists of a set of collaborating classes that clarify the overall design, class dependencies and distribution of responsibilities between them. These classes are usually abstract classes that leave implementation details to subclasses. Different subclasses implement different variants of the abstract design. Users of a framework supply it with subclasses to embed their application specific functionality into the framework.

To actually start up a system, objects of the user supplied subclasses have to be created and thus have to be named. Most current application frameworks rely on so called factory methods: These are operations whose sole purpose is to create an object of a related class specialized to fit the current class. The first application object uses factory methods to create the whole layer of user provided application objects.

Factory methods aren't essential because they are purely administrative. Their impact is even negative, because they open up class trees by explicitly naming concrete subclasses. Often a new factory method requires a new subclass to be written! However, the framework uses objects returned from factory methods under their abstract superclass's interface only. Thus, it should be the framework that retrieves user provided classes and creates objects of them.

This can be achieved by encapsulating class trees as presented in this paper. Class trees are hidden behind their abstract superclasses which serve as their interface. Clients use *class specifications* to retrieve classes (*class retrieval*) and create objects (*late* and *pattern creation*) from the encapsulated class tree. The specification mechanism is capable of dealing with hidden dependencies between subclasses.

[1] **Dirk Riehle. "How and Why to Encapsulate Class Trees." OOPSLA '95, *Conf. Proceedings.* 14 pages.**

[2] **Now at: UBILAB, Union Bank of Switzerland. CH-8021 Zürich, Switzerland. riehle@ubilab.ubs.ch**

Thus, it can be used to create several related objects using their abstract superclasses only.

Class specifications are built from clauses, which represent class properties as lightweight objects. These specifications are object-oriented right from the beginning and need no additional language or tool support. The strategies to lookup classes are so fast (in the order of factory methods) that the concepts can be incorporated into basic framework design.

The benefits of encapsulating class trees, be it for single trees or full fledged abstract designs of collaborating and dependent classes, are manifold:

- If provided with the proper data for class specifications, less code needs to be written. For example, the user is freed from specializing factory methods.
- Users can focus on the relevant abstraction and the specification of needed functionality. They don't have to deal with the implementation of an abstract design by naming specific subclasses.
- It is easier to change an encapsulated class tree, because its class names and class tree structure are hidden. Clients that work with abstract classes only are not affected by changes to the tree's structure.
- Classes can be plugged into the tree and removed with only local consequences. This eases evolution and configuration of system variants.

Encapsulating class trees makes it easier to use application frameworks, to adapt different functionality for system variants and to evolve class trees.

The next chapter contrasts this paper with a previous OOPSLA paper. Chapter 3 introduces an example framework and discusses the three main concepts needed to encapsulate class trees. Chapter 4 discusses how to represent class properties and specifications in an object-oriented way. Chapter 5 then gives concise definitions of the three main concepts from chapter 3. Chapter 6 discusses the impact of applying the techniques on both a technical and conceptual level. Chapter 7 presents related work and chapter 8 rounds up the paper with conclusions and an outlook on future work.

2 Related Work on Class Hiding

In an OOPSLA '94 paper [LS94], Lortz and Shin present a rationale for class hiding: An abstract service class provides the interface for clients but leaves open implementation details. They are filled out by concrete subclasses each with slightly differing semantics. Examples of class semantics are performance characteristics, persistence (yes/no), taking advantage of special knowledge, implementation strategies etc.

Clients of the abstract service don't know the subclasses by name. A client that wishes to create an object of the abstract service class specifies a contract for it. A contract expresses the required properties as strings consisting of keywords and other expressions, for example "persistent; range-checked; sparse; size=1000." An object returned from the creation process is guaranteed to satisfy these requirements.

The client calls a special operation of the abstract service class in order to create an object and provides it with the contract. The abstract service class holds a list of exemplars each of them representing a subclass. An exemplar is a regular instance of its class that serves as a substitute for it in languages that don't make classes first class objects, for example C++ or Eiffel. The abstract service class asks each exemplar whether it fits the given contract. If so, it is cloned and returned. Thus, the subclasses are hidden from the client though eventually objects of them are created.

Lortz and Shin's work can be interpreted to contribute to three different areas of class hiding. Their approach supports the architectural abstraction of a type with several hidden implementations, provides a string based specification mechanism and presents exemplar based programming as a design that makes class hiding possible.

This paper goes some significant steps further in all three areas. It shows that not only implementations of isolated abstract service classes can be hidden from the client but implementations of *any abstract design* consisting of collaborating and dependent classes.

It further shows how a standard metalevel architectures can be extended to support the needed specification mechanism. The specification mechanism is object-oriented and thus directly supported by the underlying language. Class lookup for given specifications can be performed by simple table lookups.

3 Overview of Concepts

Figure 1 shows a graphical editor that serves as an example throughout the paper. The editor is used to draw and arrange graphical objects. They are created from the toolbox on the left side of the window.

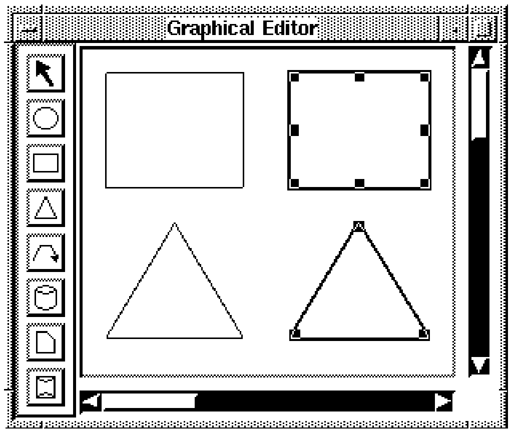

Figure 1: The graphical editor shows a toolbox of graphical objects classes on the left and some instances of them on the canvas.

The editor is built from the framework shown in figure 2. The three classes are sufficient to reveal all the relevant details of the concepts to be discussed. The figure shows a bold line that separates application specific subclasses from the framework classes. Application specific classes are, for example, rectangle and triangle classes as shown in figure 1 and resizer classes which are wrapped around a graphical object both visually and logically in order to resize it.

Figure 3 shows four of the application specific subclasses of the editor in figure 1. It is these classes that make the abstract design of figure 2 concrete and let the editor create actual graphical objects. However, from the editor's point of view it is irrelevant whether it is dealing with rectangles and triangles or symbols for resistors and capacitors. It only relies on the abstract classes of figure 2.

The graphical notation of this paper is based on OMT [RBP+91, Rum95]. Classes are drawn as rectangles and use-relationships are drawn as arrows. `GraphResizer` is a subclass of `GraphObject`. This is shown by the triangle symbol on the link connecting both classes. Objects are depicted as rounded rectangles.

The next subsections show how the editor class of the framework works with the user-provided application classes without statically referencing them.

- The editor will retrieve the classes `Rectangle` and `Triangle` from the `GraphObject` class tree to build the toolbox (class retrieval, 3.1).
- The editor will create objects of the hidden graphical object classes using a simple class specification (late creation, 3.2).

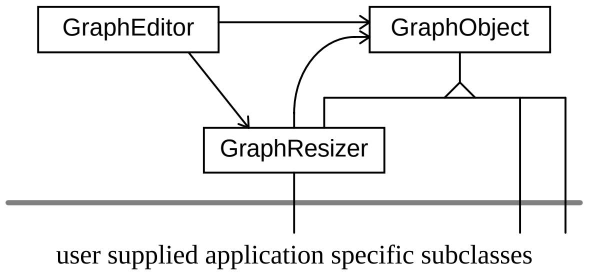

Figure 2: A simple framework for the example consisting of three dependent but abstract classes. The graphical editor class works with graphical objects and resizers.

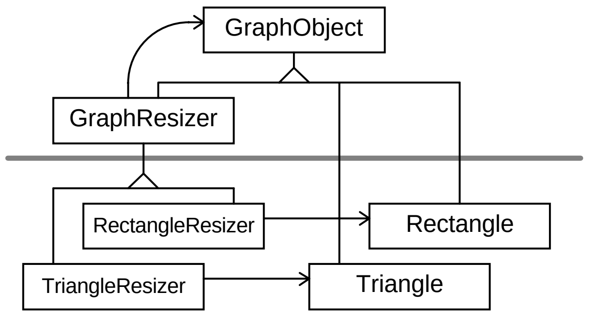

Figure 3: Excerpt from an application design. The abstract classes have been supplemented with concrete classes so it is possible to actually create a running system.

- The editor will use the same service and create a resizer for a graphical object based on an enhanced specification (pattern creation, 3.3).

The editor performs these tasks using the abstract superclasses `GraphObject` and `GraphResizer` only. Thus, the class tree is encapsulated. No class outside the class tree is granted access to its internal structure and no access is needed. The only type information available to clients outside the class tree is the one offered by the two superclasses.

3.1 Example of Class Retrieval

This section introduces the first one of two techniques presented in this paper that let clients retrieve classes from a class tree. Here, the class tree is traversed and each class is matched against a specification.

During startup time, the editor builds the toolbox of available graphical objects classes. In order to do this, it collects all classes which are capable of creating graphical objects from the class tree.

A class, as treated here, is not just a template to create instances from but an object in its own right. All classes are connected with each other according to their inheritance relationship. Thus, they form a class tree (in case of single inheritance only) or a class graph (in case of multiple inheritance). For convenience I'll stick to the notion of class tree though all concepts work with a class graph as well.

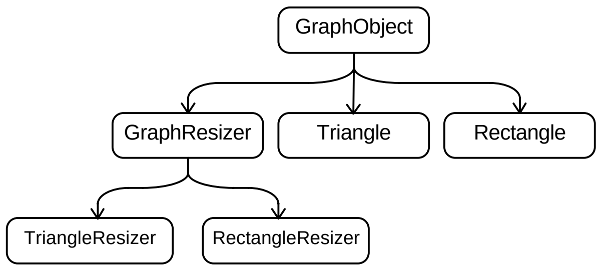

Figure 4: Each class holds a list of its subclasses. The class tree can be traversed at runtime and each node, that is a class, can be matched against a given specification.

A class tree can be traversed using standard traversal strategies. Each tree node, that is a class, is an instance of the same metaobject. Figure 4 shows the class tree for the framework example of figure 3, now depicted as an object tree.

A client can traverse the class tree and collect all classes within that tree. It doesn't have to make assumptions about class names, position within the class tree or the interfaces of their instances. The editor, for example, might traverse the tree starting with `GraphObject`, put every class into a list and build the toolbox by creating a button for each class in the list.

However, not all classes in the class tree are suitable to create graphical objects. The editor must sort out abstract classes and resizer classes, because they are not regular concrete graphical object classes. In order to do this, the graphical editor matches each class against a *specification* for concrete graphical object classes. From the class tree shown in figure 4, only `Triangle` and `Rectangle` are classes of interest with respect to the toolbox.

A specification for the given case is expressed as a simple object containing two flags: `IsAbstractClass` and `IsGraphObject`. The first flag denotes that a class is abstract (or is not, if the flag is set to false) and the second flag denotes that a class is a standalone graphical object class that might appear on a canvas. This is a simple specification sufficient for the moment. The next chapter elaborates on the notion of specification.

The editor matches each class of the graphical object class tree against the specification. Only the rectangle and triangle classes (and some more concrete graphical object classes not shown in the figures) respond positively and show up eventually in the toolbox. The next chapter discusses how to supply a class with its semantics and how to match it against a specification.

The class tree traversal and matching process is called *class retrieval*, because a number of classes adhering to a given specification are retrieved and returned to the client. It can be factored out as a service `GraphObject` offers to its clients, so that the editor hasn't got to do the traversal itself (and is prevented from making any assumptions about specific classes as well). This service can be called from the framework classes without help from subclasses, so no factory method is needed.

3.2 Example of Late Creation

This section presents a second technique to lookup a class based on a given specification. It relies on prebuilt tables that are indexed using a lookup index calculated from the specification.

If the user presses a button for a new graphical object, the editor has to create an instance of the class associated with the button. Let's assume, for the example's sake, that the editor stored only a unique identification for the class instead of the class itself. This unique identifcation, the class id, serves as a simple but effective specification of a class.

The editor now has to create an object of the class denoted by this id. This is the same task that is performed when reading objects from a stream (a file or a network link). An id appearing in the stream has to be mapped to a class so that an object can be created.

The specification is even simpler than the previous one: It is an object containing the id only. The editor might again traverse the class tree and match each class against the specification. However, this is a time consuming process. As an alternative, the editor uses a prebuilt table which is indexed by the class id. This table has been built in advance using the inverse mapping from class to id, so that the editor can use an id to lookup the corresponding class. Such a table can be built for each kind of specification, some more of which are discussed later on.

The editor looks up the class using this table and creates an object. This is called *late creation*, because it resembles the notion of late binding: The class of a new object that is to be created is determined at runtime. Late creation can also be offered as an operation of class `GraphObject` to its clients. It expects a specification and returns a new instance of a class matching the given specification.

3.3 Example of Pattern Creation

The basic concepts (class retrieval and late creation) do not only work with single class trees but with any abstract design as well. An abstract design consists of a number of abstract collaborating classes. Their implementation by subclasses often introduces hidden dependencies, for example through covariant redefinition of operations, that are not visible outside the class tree.

An example of a simple abstract design is the interplay of the classes `GraphResizer` and `GraphObject` in figure 2. A resizer wraps a graphical object both visually and logically. It draws a resize decoration around a graphical object on the canvas and it represents the graphical object to the editor. Figure 5 and 6 show a triangle resizer wrapping a triangle. A user of the editor may resize an object only by using a resizer.

A resizer class is used to factor out the resizing functionality from the class itself. Each graphical object class might have different resizing behavior, so for each graphical object class there is a special

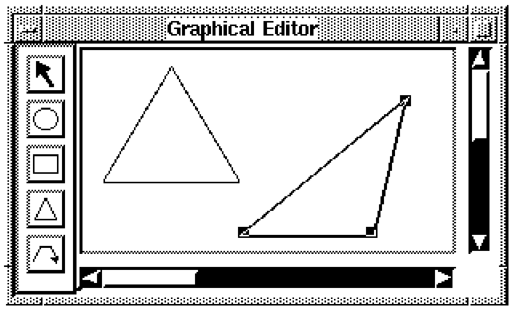

Figure 5: The user just resized the right one of the two triangles. In order to do this, the editor created a resizer object and delegated the task of resizing to it.

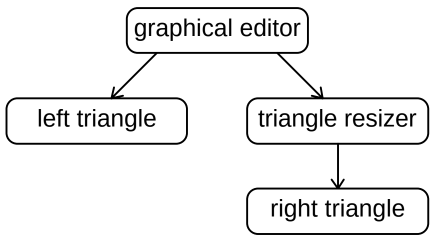

Figure 6: Object diagram of the editor in figure 5. From the editors point of view, all objects have the static type `GraphObject` or `GraphResizer`.

graphical resizer class. A triangle can only be resized by a triangle resizer. If a resizer receives an instance of a wrong class it throws an exception (covariant redefinition and design by contract [Mey91, Mey92]).

If the user wants to resize a graphical object, the editor has to find the matching resizer class. It is not always feasible to make each graphical object return its resizer, because changing requirements might force us to continuously enhance the `GraphObject` interface, just because there might be movers, draggers, iconifiers and other application-specific wrappers. This is only possible if we have source code access to the framework.

It is better to create an instance of a `GraphResizer` subclass *dependent on a given graphical object*. Thus, the editor creates a *dependency clause*, which is a certain type of specification that refers to the current computing context. This specification is an object consisting of a reference to the graphical object class the new resizer has to fit. When matched against it, each resizer class checks whether it has been designed to work with the graphical object class. If the specification contains the triangle class, only the triangle resizer class matches it.

This process is inherently the same as in late creation and thus can be performed by the same service. Only the specifications are of a more enhanced type. The process is called *pattern creation*, because it deals with setting up several related objects using only their abstract design. These kinds of designs have been called object behavioral design patterns in [GHJV93].

4 Metalevel Architecture

This chapter discusses the metalevel architecture of the approach. It is used to manage class properties and support their specification. Class semantics as well as specifications are represented by clauses. Each clause represents a specific aspect of a class's semantics and makes it a first class object.

The metalevel approach discussed here is only one possible implementation technique to achieve the overall goal of encapsulating class trees. It was chosen, because it is easy to implement and provides a clean and simple model. Essentially, it is only expected that there is some kind of representation of classes as objects, be it class objects, exemplars or prototypes.

4.1 Class Semantics

A *clause* makes an atomic statement about a class revealing true or false. It is an instance of a clause class which represents a specific aspect of another class's semantics as an object. Examples given so far are *property clauses* which consist of flags for simple class properties, for example whether a class is abstract or not. *Identification clauses* denote a single class unambiguously by giving a unique identifier. *Dependency clauses* refer to other classes in order to make a dependency relationship explicit, for example between a resizer and a graphical object class. Dependency clauses are often identification clauses (expressed by subclassing, see figure 7).

Clauses are used for two different but complementary purposes. First, a client of a class tree uses clauses to build a *specification* for a class to be retrieved from the tree. Second, a class holds a list of clauses that are said to *represent its semantics* as simple objects.

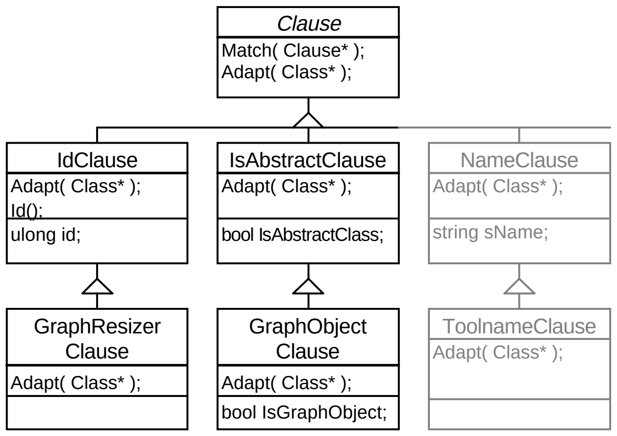

Figure 7: The `Clause` class tree for the framework example shows the discussed clauses on the left with bold lines and characters.

Both specifications and class semantics rely on clauses. Thus, a specification can easily be matched with a class, because its basic constituents can be compared in a simple way (by default an equality check).

A *specification* is a formula from propositional calculus with clauses as its atomic constituents. Propositional calculus is used for intensional class semantics. For extensional semantics, considering not only the class but all its instances too, predicate calculus has to be used. However, propositional calculus is not particularly interesting for the current discussion, so the rest of the chapter focuses on clauses and assumes a specification to consist of a single clause.

A client that creates a clause to be part of a specification provides it with all relevant data. The editor, creating a `GraphObjectClause` instance, provides it with flags that indicate whether the graphical object class has to be abstract and whether it has to be a standalone graphical object class.

A class holds a list of clauses to represent its semantics as first class objects. `GraphObject` holds instances of the clause classes `IdClause`, `IsAbstractClause`, `GraphObjectClause` and others not mentioned here as well. Each clause instance makes an atomic statement about the class it is held by. Thus, a clause is an effective representation of a specific meta information about a class.

In order to achieve this, the clause has been created with the class as its initial parameter. The clause analyzes the class and extracts those data that are relevant for it. Thus, a clause presents a specific view on a class and makes properties explicit that have only implicitly been available.

The class itself offers the data needed by a clause. For example, each resizer class offers an additional operation that returns the graphical object class the resizer class has been designed to work with. A graphical resizer clause asks its resizer class about the graphical object class so that it can be matched against a similar clause from a specification later on.

At first glance this seems to be a strange turn: Formal semantics of a class are made explicit through clauses held by that class, but the clauses extract the needed data for these semantics from the class itself. It surely is possible to retrieve the data needed by the clause instances from external databases, macros or other textual specifications. The object-oriented way proposed here has the advantage that all data is immediately type checked, resides at the class where it belongs and makes classes provide default semantics.

This might change in future versions with more tool support than just the underlying language. However, in our experience it has shown to work satisfactorily and to scale without problems. So, there is currently little need for more advanced specification techniques.

The matching process is reduced to matching the clauses and evaluating the formula. Matching the clauses of a specification is done by comparing them with their counterpart (if available) in the class's clause list. This is by default an equality check. The responsible operation can be specialized to fit more peculiar comparison semantics.

Clause classes are introduced for a specific class in the overall system's class tree. Due to the substitutability principle [Lis88], subclasses of that class hold instances of the clause class too, whereas superclasses don't. The general root class of the system, `Object`, holds instances of `IsAbstractClause`, `IdClause` etc., but no instance of `GraphObjectClause`. The last clause has been introduced for `GraphObject` and all its subclasses. `GraphObject`, in turn, doesn't hold a `GraphResizerClause` instance, but `GraphResizer` does, and so on.

A class's type interface, operation signatures and semantics, pre- and postconditions, invariants and history properties [Mey91, LW93] can be represented by clauses, too. The clause classes proposed here are mainly pragmatic: They have been introduced because it could be foreseen that a class tree will be asked specific questions regarding properties represented by these clauses. The metalevel architecture discussed in the next subsection shows that new clause classes can be introduced to already existing and encapsulated class trees solely based on their interfaces.

4.2 Metaobject Extensions

Obviously, a class has to be extended so that it can hold a list of clauses and provides operations to match a specification against these clauses. This is a straightforward task not discussed further. However, it has been left unclear how clause classes come into the system and are distributed to their classes.

Some kind of managing facility has to exist that keeps track of classes and the clause classes associated with them. This can be the metaclass or a separate object dedicated to this single task. The managing facility associates clause classes with classes and creates clause instances. The instances get adapted to their specific classes, that is a certain root class for the given clause class and all its subclasses.

5 Class Tree Encapsulation

The overall idea is simple: Class trees get encapsulated by granting access only through class specifications. Class semantics are split up into several distinct properties each of them represented by a clause class. Instances of these clause classes are used to both specify the semantics of a certain class and to build specifications for classes of interest.

However, there are at least three different ways of using this idea (class retrieval, late creation and pattern creation) each serving a different pragmatic purpose. A client that uses all of them can successfully restrict its knowledge about a design to its abstract superclasses. Thus, the subclasses as implementations of the design are hidden and the class trees are encapsulated. The abstract superclasses serve as the *interface classes* of the class tree.

The next three subsections give concise definitions of the three basic concepts and the fourth subsection gives a conclusion about client specific encapsulation of class trees.

5.1 Class Retrieval

Class retrieval of a set of classes is the process of looking up all classes which are subclasses of a specific root class and adhere to a given specification.

A convenient way to encapsulate class retrieval is to make each class offer a service operation that performs this task. A class that returns a set of classes is called the *root class* for it. Each object of a class in the set is guaranteed to support the interface declared by the root class. A compiler can statically rely on this. In addition, each class matches the specification the operation has been supplied with. For the client, the root class takes over the role of the interface class of the class tree.

The service operation hides how the lookup is performed. The two straightforward techniques presented in this paper are either traversing the class tree or using prebuilt tables to perform a table lookup. Specifications for class retrieval usually denote a set of classes instead of single class.

Clients use the set of classes the root class returns to gain knowledge about available functionality, for example before presenting a menu item to a user. They can use it to decide on how to proceed further.

5.2 Late Creation

Late creation of an object is the process of creating an instance of a class by naming only an abstract superclass of that class and giving a specification.

Late creation can be offered as another service operation of each class. It accepts a specification and returns an object. The object (or null, if no object was created) conforms to the interface defined by the root class and provides the specified properties. Again, a compiler can statically rely on the interface.

It usually is implemented by looking up the actual class for the new object in a table made for the specification type. These tables can be prebuilt, for example for all specifications that consist of a single clause only. Since there is a possibly large number of different specifications, one has to make a choice in the beginning and create further tables on demand.

If such a clause denotes a class unambiguously, the table can be built easily. It is calculated using the inverse mapping from class to unambiguous identification from the clause instance for that class. If the specification is ambiguous, an entry in the table reveals the set of equivalent classes with respect to the given specification. Then, further information is needed. For example, the client may specify, whether it is interested in the most general or most specialized class. This can be done by imposing different *strategies* or *function closures* on the class tree traversal [GHJV95, Küh95] for a depth or breadth first traversal.

However, no general solution to resolve ambiguous specifications exists and the client should make no more assumptions than what it has specified.

5.3 Pattern Creation

Pattern creation is a special case of late creation for which the specification is based on clauses that refer to the current computing context.

In object-oriented systems very often not only a single class but a number of collaborating classes are subclassed. The collaborating classes implement a new variant of an abstract design. They take advantage of being subclassed in concert by directly referring to the other classes' interfaces. Thus, objects of these classes can only be used together. This dependency is secretly introduced behind an abstract design. It is not visible to clients of the abstract superclasses. This has to be taken into account, if a new object of a class of the abstract design is to be created using late creation. Thus, the specification has to refer to the objects, that is the current computing context, the new object has to fit.

A clause *refers to the current computing context if it is built on behalf of objects of that context*, for example by referencing the objects' classes. Referring to the computing context lets a clause carry data about class dependencies based on a current object constellation. For example, `GraphResizerClause` instances refer to the class of the graphical object the new resizer has to fit. This can be matched easily against the internal clauses of a resizer class. Only the correct resizer class responds positively.

A table can be built using the inverse mapping from graphical object class to resizer class. This table can then be indexed with the class reference (or its id or symbol) from the clause. The lookup returns returns the denoted resizer class dependent on the graphical object class in the clause.

Clauses that carry such dependency data are called *dependency clauses*. They are needed to instantiate an abstract design using only its abstract superclasses. Usually the first object of an abstract design is created using regular late creation and then the other objects are created depending on this first one using pattern creation. This is a bottom up process that ensures the instantiation of a correct implementation of a design. Finally, it is possible to selectively instantiate parts of an abstract design. This is important, because most designs interact with other designs only partially.

Thus, the general specification for a new object that has to fit a given object constellation is a conjunction of dependency clauses. Each clause identifies the class of an already existing object and its role which the new object has to match.

5.4 Clients and Encapsulation

Each of the three presented concepts makes a different contribution to encapsulate implementations of potentially any abstract design. Using all three concepts together, no creation and selection process has to know more about the intended classes and their instances than the (possibly abstract) interface the client chooses to work with.

It has to be stressed that class tree encapsulation is always done from a client's perspective. For different clients different subtrees of the overall class tree become encapsulated. For the general algorithm of object activation and passivation the relevant classes are `Object` or `Persistent`. All other classes are hidden behind them. For `GraphEditor` the classes `GraphObject` and `GraphResizer` are relevant and all subclasses are hidden behind them, too. For each graphical resizer class a specific graphical object class is relevant the interface of which it knows in detail.

Thus, no overall class tree encapsulation is achieved (which wouldn't make sense anyway) but always a client specific one.

6 Impacts and Discussion

At least three areas worth of discussing can be identified: The implementation techniques, the applicability of the techniques and the impact and consequences of class tree encapsulation.

6.1 Implementation Techniques

Metaobject protocols have been well justified for dynamically typed languages [Att93, KAR+93, KRB91]. They are used in many application frameworks for statically typed languages [CIRM93, GOP90, LVC89, WG94], because language provided features [Mey92, Str94] are often not sufficient. Ways of doing this in C++ are shown in [BKS92, Gro93, IL90, PWJ92].

The downside of an explicit metalevel architecture is some memory and management overhead. The approach proposed here increases this drawback, because memory consumption goes up due to the clauses held by each class. It depends on the type of application and its environment whether this is a problem. Today, this shouldn't be a severe problem with most applications.

Class retrieval and late creation use up more time than simple factory methods. If a class tree traversal is involved, it should carefully be analyzed whether performance problems might result. If a table lookup is sufficient, only constant time is consumed.

The described metalevel architecture integrates easily with current application frameworks, at least those mentioned above. Class retrieval and late creation will then be available for every class. Clauses and specifications are type checked because they are object-oriented right from the beginning. It is possible to introduce new explicit semantics to a class tree at any time as long as the classes provide sufficient data to derive that semantics. Tool support for specifying semantics can make any new clause class possible.

6.2 Basic Framework Applicability

Class retrieval and late creation can be applied in basic framework design. It depends on the speed requirements and memory limits whether it is feasible and sensible. In our C++ and Smalltalk frameworks [RZ95, RS95], we implemented class retrieval as a class tree traversal and late creation for specifications consisting of a single `IdClause` instance as a table lookup.

Class retrieval was never applied in time critical parts of our interactive systems. The memory overhead for representing class semantics as a set of clause instances is fixed and can be calculated in advance. It didn't lead to problems for us.

Late creation was used to pervasively replace factory methods. Late creation of an object uses up more time than a factory method call, however, the delay is constant and could be neglected for most of the situations we had to deal with.

Facilities for retrieving the data for class properties have to be introduced. This paper proposed the simple technique of providing class properties through simple access operations of a class. Clause classes are designed using these operations. Not having to specialize factory methods that were replaced by late creation makes up for the additional effort of writing these access operations. The class property access operations may be reused by new clause classes while factory methods serve only a single pupose. Moreover, the access operations provide default semantics and needn't be specialized by every new class. Thus, compared to factory methods, class retrieval and late creation reduce coding effort in the long run.

6.3 Class Tree Encapsulation

The most relevant problem of the approach is also one of its biggest strengths. The configuration of a system has now to be specified on a metalevel, for example a makefile. Classes of encapsulated class trees have to be linked explicitly, because they are no longer statically referenced by client code. Otherwise they will not be included in the resulting executable. Careful attention has to be paid in order not to forget relevant classes.

However, it is much better to specify systems on a metalevel as opposed to writing code that causes classes to be linked. This avoids system variants maintenance problems. In the long run it can be imagined that variants are specified solely on a tool or makefile level which both draw on a common pool of available classes. This is already a reality with the applications we work with: It is specified in a makefile which tools, materials and mediators are to be linked. Thus, we easily create different system variants be it for restricted use, full use or system maintenance.

The main driving principle behind the three concepts and the metalevel architecture is the notion of encapsulation and information hiding. Though the well known benefits apply, the situation is more complicated. Different clients need different knowledge about interfaces. The notion of client specific class tree encapsulation expresses this. Each client's knowledge about interfaces is restricted to those interfaces it finally chooses to work with. The three concepts of class retrieval, late creation and pattern creation hide the selection and creation process of objects. They successfully restrict a client's knowledge of an object's interface to the minimum interface possible and sensible.

The concepts ease configuration of system variants and evolution of class trees. They enhance information hiding and thus intellectual manageability. They reduce the learning effort needed to understand a framework because details like class names and class tree structure become less important. User can concentrate on the relevant abstraction they are interested in.

7 Related Work

Several aspects of the work presented can be found in related works. Coplien uses exemplar based programming as an alternative to classes (generic autonomous exemplar idiom [Cop92]) and supplies them with make operations to create objects of hidden subclasses. This achieves class hiding as described in chapter 2.

Gamma et al. use factory methods, abstract factories, builders and prototypes [GHJV95] to separate the creation processes from the clients that initiate them. Factory methods are used to create the correct objects of a dependent class, abstract factories bundle factory methods for a specific system and prototypes serve just like exemplars as a substitute for classes. Builders are objects that perform the creation of complex structures like compound documents. A builder has an abstract interface that decouples the client from the internal complexity of the structure to be built.

Chambers introduces Predicate Classes [Cha93] as a new linguistic concept for programming languages. Predicate classes are ordinary classes that have a predicate associated with it. Every object that fulfills the predicate automatically becomes an instance of that predicate class. Thus, method dispatch can be based on the state of an object. The programming language has to offer dynamic reconfiguration of class hierarchies and must be capable of initiating this task itself.

Lieberherr et al. generate programs from specifications based on propagation patterns and class dictionary graphs (adaptive programming [LSX94, LX94]). An adaptive program consists of a number of propagation patterns that specify constraints on class relationships. To generate an actual program, the propagation patterns are customized by class graphs which set up a class structure conforming to the pattern constraints. Thus, an adaptive program denotes a family of programs which are constrained by the propagation patterns. Lieberherr et al. rely on a CASE system.

General approaches to distributed environments like CORBA [OMG91, OMG92] work with interface definitions for objects to be returned from a request. Thus, interface definition languages [IDL94] and object request brokers are needed. In addition, adaptors between dynamically specified interfaces and their implementations have to be invented.

8 Outlook

Currently the two most important open issues are tool support for configuring system variants and means for embedding dynamically linked libraries. A description for class dependencies has to be introduced, for example based on [HHG90, Sha94, AG94] or parts of [SDK+95, GAO94], and supported by a tool. The tool eventually has to generate a list of classes to be delivered either statically linked or as a dynamically linked library.

Compared to object request brokers and interface definition languages the concepts proposed in this paper are quite simple. However, they can be integrated with current application frameworks and don't require a major overhead in order to do so. This easy integration with current metaobject protocols makes the concepts usable now and might eventually make them into a basic technique for framework design. The experiences with using the techniques in our frameworks are very promising: Once they were introduced, everyone wanted to use them and did so successfully. We hope to give an experience report on the wide variety of applications of class tree encapsulation in the near future.

Acknowledgments

I wish to thank Ralph Johnson and Heinz Züllighoven who both reviewed the paper and helped me define its scope in a way suitable for OOPSLA.

I would like to thank Walter Bischofberger, Bradley Edelman, Kai-Uwe Mätzel, Thomas Kofler, Peter von Savigny, Wolf Siberski and the anonymous reviewers for their supporting and critical but always helpful comments on the paper.

Bibliography

AG94 Robert Allen and David Garlan. "Formalizing Architectural Connection." ICSE-16, *Conference Proceedings*. Los Alamitos, California: IEEE Computer Society Press, 1994. 71-80.

Att93 Giuseppe Attardi. "Metaobject Programming in CLOS." *Object-Oriented Programming: The CLOS Perspective*. Edited by Andreas Paepcke. Cambrigde, Massachusetts: MIT Press, 1993. 119-131.

BKS92 Frank Buschmann, Konrad Kiefer and Michael Stal. "A Runtime Type Information System For C++." Tools-7, *Conference Proceedings*. Edited by Georg Heeg, Boris Magnusson and Bertrand Meyer. New York, London: Prentice-Hall, 1992. 265-274.

Cha93 Craig Chambers. "Predicate Classes." ECOOP '93, LNCS 707, *Conference Proceedings*. Berlin, Heidelberg: Springer-Verlag, 1993. 268-296.

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.

GAO94 David Garlan, Robert Allen and John Ockerbloom. "Exploiting Style in Architectural Design Environments." SIGSOFT '94, *Software Engineering Notes* 19, 5 (December 1994): 175-188.

GHJV93 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. "Design Patterns: Abstraction and Reuse of Object-Oriented Design." ECOOP '93, LNCS 707, *Conference Proceedings*, 1993. 406-431.

GHJV95 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 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.

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.

HHG90 Richard Helm, Ian M. Holland and Dipayan Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems."

OOPSLA ’90, *ACM SIGPLAN Notices* 25, 10 (October 1990): 169-180.

IDL94 Interface Definition Language Workshop, *ACM SIGPLAN Notices* 29, 8 (August 1994).

IL90 John A. Interrante and Mark A. Linton. “Run-Time Access to Type Information in C++.” USENIX, *Conference Proceedings 1990*. 233-240.

KAR+93 Gregor Kiczales, J. Michael Ashley, Luis H. Rodriguez Jr., Amin Vahdat and Daniel G. Bobrow. “Metaobject Protocols: Why We Want Them and What Else They Can Do.” *Object-Oriented Programming: The CLOS Perspective*. Edited by Andreas Paepcke. Cambridge, Massachusetts: MIT Press, 1993. 101-118.

KRB91 Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press, 1991.

Küh95 Thomas Kühne. “Parameterization Versus Inheritance.” TOOLS-15, *Conference Proceedings*. Edited by Christine Mingins and Bertrand Meyer. New York, London: Prentice-Hall, 1995. 235-245.

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.

LSX94 Karl J. Lieberherr, Ignacio Silva-Lepe and Cun Xiao. “Adaptive Object-Oriented Programming.” *Communications of the ACM* 37, 5 (May 1994): 94-101.

LVC89 Mark A. Linton, John M. Vlissides and Paul R. Calder. “Composing User Interfaces with InterViews.” *IEEE Computer* 22, 2 (February 1989): 8-22.

LW93 Barbara Liskov and Jeannette Wing. “A New Definition of the Subtype Relation.” ECOOP ’93, LNCS 707, *Conference Proceedings*. Berlin, Heidelberg: Springer-Verlag, 1993. 118-141.

LX94 Karl J. Lieberherr and Cun Xiao. “Customizing Adaptive Software to Object-Oriented Software Using Grammars.” *International Journal of Foundations of Computer Science* 5, 2 (1994): 179-208.

Mey91 Bertrand Meyer. “Design by Contract.” *Advances in Object-Oriented Software Engineering*. Edited by Dino Mandrioli und Bertrand Meyer. New York, London: Prentice-Hall, 1991. 1-50.

Mey92 Bertrand Meyer. *Eiffel. The Language*. New York, London: Prentice-Hall, 1992.

OMG91 Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 1.1 (OMG Document 91.12.1), 1991.

OMG92 Object Management Group. *Object Management and Architecture Guide*. 2nd Edition (OMG Document 92.11.1), 1992.

PWJ92 Peter-Alexander Pauw, Ronald Werring and Angelique Jansen. “An Operational Metadata System for C++.” Tools-8, *Conference Proceedings*. Edited by Raimund Ege, Madhu Singh and Bertrand Meyer. New York, London: Prentice-Hall, 1992. 215-223.

RBP+91 James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen. *Object-Oriented Modeling and Design*. London: Prentice-Hall, 1991.

Rum95 James Rumbaugh. “OMT: The Object Model.” *Journal of Object-Oriented Programming* 7, 8 (January 1995): 21-27.

RS95 Dirk Riehle and Martin Schnyder. “Design and Implementation of a Smalltalk Application Framework for the Tools and Materials Metaphor.” *UBILAB Technical Report No. 95.6.1*. In preparation.

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.

SDK+95 Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M.Young and Gregory Zelesnik. “Abstractions for Software Architecture and Tools to Support Them.” *IEEE Transactions on Software Engineering*. To appear.

Sha94 Mary Shaw. “Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status.” *Proceedings of the Workshop on Studies of Software Design*. New York: Springer-Verlag, 1994.

Str94 Bjarne Stroustrup. *The Design and Evolution of C++*. Reading, Massachusetts: Addison-Wesley, 1994.

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: Universitätsverlag Konstanz, 1994. 66-92.