USING, REUSING AND DESCRIBING OBJECT-ORIENTED FRAMEWORKS

by
Richard Lajoie

School of Computer Science
McGill University, Montreal

October 1993

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Software reuse is concerned with capturing software components in some form and then applying them to the construction of another application. The ultimate goals of software reuse are: to improve the quality of software produced; reduce the costs of software development; and increase the productivity of software developers. However, the present situation, and the concern amongst researchers, is that software reuse is not living up to its original expectations [FBPD+91].

Although there have not been very many scientific studies that validate the claim, it is nonetheless a strongly held belief among researchers and developers that object-oriented software offers great potential in terms of software reuse [Mey88]. Much as object-oriented programming allows for the creation of more reusable components, it is the reuse of the design of an application that is most promising for attaining the goals of reusability. For this reason, our work concentrates on *application frameworks*, an important object-oriented technique to facilitate design-level reuse.

We have spent the last year developing a fairly large object-oriented application called *Macrotec*, using an application framework called ET++ [WGM89]. Throughout this paper, we will refer to our experience, and in-experience, in working with application frameworks. In particular, we will introduce the different levels of reuse we have identified when developing from application frameworks. We address the present lack of adequate design representations by introducing a new technique for the representation of abstract designs. Also, a new approach to documenting application frameworks is presented, complementing the proposed representation of abstract designs.

We strongly believe that application frameworks have the potential of drastically improving the current reuse crisis. We sincerely hope that our work will become a

first step in solving the current lack of knowledge in how to use and reuse application frameworks effectively.

Abstrait

La réutilisation de logiciels se penche sur la saisi d'éléments de logiciels sous une forme quelconque pour ensuite les appliquer à la construction de logiciels servants à une autre application. Les buts ultimes de la réutilisation de logiciels sont: d'améliorer la qualité du logiciel produit; de réduire les coûts de développment de logiciels; et, d'augmenter la productivité des développeurs de logiciels. Toutefois, la situation présente qui préoccupe les chercheurs, est que la réutilisation des logiciels ne produit pas les résultats escomptés [FBPD+91].

Quoiqu'il n'y a pas eu beaucoup d'études scientifiques qui ont pu valider cette assertion, il n'en demeure pas moins que les chercheurs et les développeurs croient que le logiciel orienté objet offre beaucoup de potentiel en termes de la réutilisation de logiciels [Mey88]. Même si la programmation orienté objet permet la création d'éléments beaucoup plus réutilisables, c'est cette réutilisation du design d'une application qui demeure la plus prometteuse dans la course d'obtention de la réutilisation. Ainsi, cet ouvrage porte sur les librairies à une racine, une technique orienté objet importante, utilisée afin de faciliter la réutilisation au niveau du design.

Nous avons passé la dernière année à décelopper une application orienté objet appelé *Macrotec*, en utilisant une librairie à une racine, appelée ET++ [WGM89]. Nous allons donc, tout au long de cet ouvrage, référer à notre expérience avec les librairies à une racine, et par surcroit, à notre ignorance. Nous soulignons l'absence de designs représentatifs en introduisant une nouvelle technique pour la représentation de designs abstraits. Aussi, nous présentons une nouvelle approche pour l'application de librairies à une racine. Celle-ci s'ajoutera à la représentation de designs abstraits proposée.

Nous croyons fermement que la librairie à une racine a le potentiel d'améliorer d'une façon draconienne la crise de la réutilisation. Nous souhaitons que ce travail

devienne une première étape dans le processus de solution du comment de l'utilisation et de la réutilisation efficiente de librairies à une racine.

Acknowledgements

I thank my friend, supervisor, Rudulf Keller, for his tremendous guidance and patience during my studies at McGill University. I learned a great deal from our many brainstorming and debugging sessions. I was amazed by both his technical skills and theoretical knowledge in the object-oriented domain. He invested his most valuable resource on my behalf, his time. I am also grateful to my co-supervisor, Prof. Nazim Madhavji, for teaching me the necessary software process skills required for the successful development of a complex system such as *Macrotec*.

My close friend and Tiger brother, Xijin Shen, was the best software development partner I have ever had the privilege of coding with. We motivated each other throughout many exceptionally long coding sessions. Thank you Xijin for being so patient. I can only hope that I will once again have the opportunity to work with you in the near future.

I would like to thank the proof readers for having invested so much time and effort. Your comments improved this paper many fold. Rudolf gave himself heart and sole to the proof reading process and I paid for it dearly (haha). My close friend, Stephane Alarie, was always there when I needed to take my mind off my work. We shared in many PP sessions which always did the trick in getting me back into a working mood. A big thanks to Guylaine and Ron for their help in translating the abstract (actually they did it all, all I had to do was learn how to use the fax machine).

A sincere "merci" goes out to my special friend Isabelle. She is the most understanding and patient person I know. Without her presence, and her ability to make me laugh, the past two years would have been unbearable.

I thank my parents for their support. My mother for her pampering, and my father for his ability to put today's 'crisis' in perspective. My sister Guylaine encouraged me

throughout my studies and was always ready to support her little brother. Thanks Guy!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Frameworks and the Case for Reuse

Let's face it, we, software engineers, have been asked to implement, manage, and comprehend highly complex systems with unrealistic, in fact ridiculous, expectations. In the past, we did not realize how outlandish the development of such systems really was, however the advent of software reuse has opened the eyes of many. We have now realized, that through reuse, it is in fact possible to successfully build those complex systems, and to build them faster and better than ever before. Brooks has stated that software reuse is an area where the greatest productivity results can be achieved because reuse addresses the "essence", as opposed to the "accidents" of the development problem [Bro87]. It is however a mistake to assume that reuse does not pose new challenges. According to Freeman, the state-of-the-practice of reuse is embarrassing [Fre87]. The present situation, and concern amongst the researchers, is that software reuse is not living up to its original expectations.

Implementors of complex systems are generally reluctant to engage in reuse and redesign[1]. This has mainly been attributed to an aversion to "not invented by me code" and a lack of technical support for the reuse of design. It is very difficult to reconstruct the design principles that underlie a given module by reading the code for it. In fact, the comprehension of unfamiliar code is, in my experience, one of the

[1]The managerial, cultural and organizational issues involved in software reuse [PD91b] are beyond the scope of this work. We do, however, consider them equally important.

most underestimated tasks in software engineering. Meyer [Mey87] and many others claim that object-oriented design is the most promising technique for attaining the goals of reusability. In particular, object-oriented frameworks have the potential to significantly facilitate design-level reuse [WBJ90, Kru92]. However, little is known how to use and reuse them effectively.

1.2 Research Problems and Solution Approach

Wirfs-Brock and Johnson [WBJ90] identify three main research areas related to frameworks. The first is designing frameworks: what are the characteristics of a good framework and how is one designed? The second is using frameworks: how does one configure a particular application based on a framework? The third is describing frameworks: what notation is needed, other than that applicable to object-oriented design in general?

Although the first issue mentioned is highly interesting, it is beyond our present knowledge in the domain. We therefore leave it to the experts, i.e. those few who have experienced, over many years, the development of object-oriented frameworks. In this work, we shall provide our insights and experience in the last two issues mentioned. In particular, we will address the present lack of adequate representations to facilitate the reuse of design. In fact, we go one step further and suggest that there are several levels of framework reuse depending on the level of software abstraction. The higher the level of abstraction, the greater is the potential for reuse. Also, because of the sheer size (a large collection of classes) and complexity of frameworks, we will attempt to describe them in a much more practical manner, one encouraging their reuse.

1.3 Contributions

The major contributions of this research are:

- the introduction and description of different reuse levels at different levels of abstraction;

- the identification of framework micro-architectures as reusable design components;
- a new technique for the representation of abstract designs;
- a new approach to documenting frameworks which compliments the proposed representation of abstract designs and thus promotes framework use;
- The validation of these concepts with the development of the Macrotec Toolset [KLO+93a, KLO+93b].

The ultimate goal of this research is to incite framework designers to adopt our reuse philosophy and description techniques thus providing application developers with a more comprehensible and thus reusable framework. In turn, application developers, also adopting our suggestions, will produce highly flexible, thus easily maintainable and reusable systems.

1.4 Outline

This work is organized as follows: Chapter 2 begins with a discussion about the object-oriented paradigm and its support for reuse. The same chapter then introduces object-oriented frameworks as well as the different reuse levels we have identified. Chapter 3 describes the *Macrotec* toolset, a system we have developed using the ET++ object-oriented framework. The ET++ framework itself is then briefly introduced. Chapter 4 treats design and its reuse in frameworks. It begins with a discussion of why design has long been considered a difficult task. Then, the two lowest levels of reuse, reuse-in-the-small and reuse-in-the-medium, are described with a concentration on the latter, specifically the reuse of micro-architectures. In chapter 5 we briefly introduce the highest level of reuse, i.e. the reuse of "black box" applications. Chapter 6 identifies important framework learnability issues we deemed important for successful reuse. We also present a known framework description technique we have adapted to supplement the design reuse techniques of chapter 4. Finally, chapter 7 summarizes our work and lists some of the advantages of using frameworks we have experienced in the development of *Macrotec*. We then mention possible improvements/extensions to our presented framework description techniques. We conclude with a few final remarks.

Chapter 2

Background

This chapter will review the object-oriented paradigm in terms of reuse potential. The object-oriented technique of frameworks and their inherent levels of reuse will be discussed.

2.1 Object-Oriented Programming and Reuse

This section will describe why the object-oriented paradigm has been touted as the answer to the reuse crisis. We shall begin by describing the differences between object-oriented and procedural solutions in an attempt to understand why the procedural paradigm has not been as successful with respect to software reuse. We will then summarize and comment on an empirical study of software reuse performed at *Virginia Tech* [LHKS91].

2.1.1 The Object-Oriented Paradigm

There are several important characteristics in the object-oriented paradigm which permit flexibility in defining and composing reusable components. Indeed, object-oriented languages are required to support the four concepts of data abstraction, information hiding, inheritance, and polymorphism [ES92]. All of these concepts

facilitate reuse, yet at different levels and to various extent. Inheritance and polymorphism are the two concepts which set object-oriented languages apart from other high-level procedural programming languages. Languages like Modula-2 [Wir85] and Ada [Bar84] have the feature of a module (Modula-2) and a package (Ada), allowing them to support data abstraction and information hiding. In terms of reusability, this is definitely a step in the right direction. However, it is limited, because the abstractions (types, modules, packages) are not easily extensible, and because it is difficult to capture common features between modules.

Consider for example the following definition of a type **Animal**. Assume for the moment that the system must support tigers and house cats. Also assume a type **Location**.

Under such circumstances you could have the following (in C++):

```
enum species{Tiger,HouseCat};

class Animal{
  Location position;
  species kind;

public:
  Location where() { return position; }
  void moved(Location here) { position = here; }
  void feed();

};
```

The enumerated type, **species**, is used by **feed()** to determine which animal must be fed. The function feed() could be defined as:

```
void Animal::feed()
{
  switch(kind){
  case Tiger:
      // feed the big fellow steak
```

```
            break;
        case HouseCat:
            // feed the cute little thing milk
        }
    }
```

The type **Animal** as defined above is not flexible enough. It requires the function feed() to know all the animal types. The problem is that we cannot distinguish between the general properties (all animals have a location and they move) and those specific to each animal (feeding habits). Adding new animals will typically require several functions to be modified, risking the introduction of errors. The module is therefore said to be representation dependent.

There are two additional features unique to the object-oriented paradigm, which in combination with encapsulation and data abstraction, permit the above flexibility. They are: inheritance; and, polymorphism.

Inheritance

Inheritance permits a type (in object-oriented terms: class) to inherit operations as well as internal structures (internal data members and methods) from another class, a super class. The term inherit in the previous sentence could just as well have been replaced with "make use of". A class (base class) inherits from its super class and hence may make use of certain functionality (methods) and/or data (data members) defined within that super class. A super class may in turn itself be a base class with respect to another (higher level) super class. A base class inherits from its parent class, its parent's parent class and so on. Figure 1 depicts a sample inheritance hierarchy for the class animal.

A class may add to the operations it inherits or redefine inherited operations. It may not restrict inheritance by choosing not to inherit certain operations from its parents. A base class is in fact a specialization of its parent class. All the inherited data and operations are free i.e. they have already been implemented and are stable relieving the base class designer from rewriting the inherited operations. The designer is required to code only those operations which are different from that of its parents hence adopting a style of programming called *programming-by-difference*. Another

obvious benefit is that modifications (specializations) are made in the base class leaving the original inherited code intact.

Inheritance is one of the major extensions to abstract data types provided by the object-oriented paradigm. In summary then, object-oriented programming is programming using inheritance, and data abstraction is programming using user-defined types [ES92].

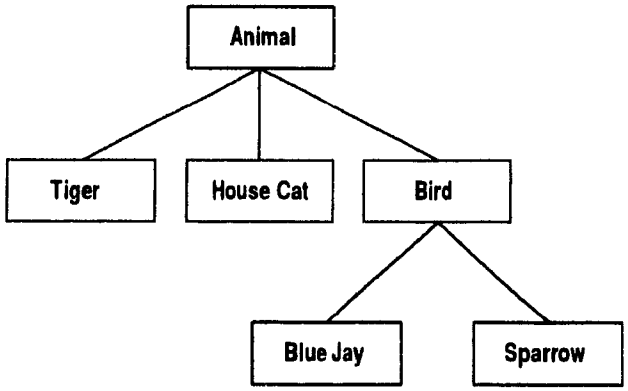


Figure 1: A sample inheritance hierarchy.

Polymorphism

The second major extension to abstract data types that comes with the object-oriented paradigm is polymorphism. Polymorphism is the notion that a procedure can be invoked for an object without knowing that object's exact type. Polymorphism is supported by the late binding of function calls. The C++ mechanism that provides late binding is called a *virtual method* ("virtual" is the Simula and C++ term for, may be redefined later in a class derived from this one) [Str88]. Each instance of a class that defines or inherits virtual functions has a pointer to a virtual function table, called a *vtable*. When a virtual method is invoked for an object, the appropriate method is invoked by retrieving the address of the function from the vtable [Lip92]. Polymorphism allows software to be more general (applicable to more kinds of data) and extensible (applicable to as yet unspecified data).

main() provides a simple demonstration of how the features of inheritance and polymorphism may be used;

```
class Animal
{
  Location position;
...
 public:
   Location where()
    { return position; }
   void moved(Location here)
    { position = here; }
   virtual void feed();
...
}
 main()
  {
    Location place;
    HouseCat *KittyCat;
    KittyCat->moved(place);      //will call moved() of class
                                 //HouseCat which is however
                                 //general, i.e., defined
                                 //in its superclass Animal
    feeder(KittyCat);
  }

 function feeder(Animal *kind)
  {
   ...
   kind->feed();  //will call feed() of class HouseCat due
                  //to polymorphism.
   ...
  }
```

```
class HouseCat : public Animal
{

 ...
  public:
   void feed()
    { //feed pussy cat 2% milk}
}
```

Scope resolution allows the development of a hierarchy of classes in which a derived class can inherit a common method from its parent, in this case methods **where()** and **moved()**. A common message, for example, "kind->feed()", can invoke different methods in a class hierarchy, depending on the type of **kind**. This feature is what we

have described as polymorphism.

The concepts of polymorphism and inheritance thus support flexibility and extensibility. They are inherent in the object-oriented paradigm and are the key concepts for promoting software reuse.

2.1.2 An Empirical Study of Software Reuse

Now that we agree upon the object-oriented paradigm's affinity for reuse, the next logical step is to determine the effects of this paradigm on software reuse.

A study, recently completed at *Virginia Tech* [LHKS91], measured the relative impact of a procedural language and an object-oriented language on software reuse.

The Experiment

The experiment was conducted on a target system whose implementation involved a variety of programming techniques. These techniques were drawn from the "employee management" and "business management" domains and included data management, numerical processing, and graphics.

In the experiment, two sets of reusable code components were made available to the subjects implementing the target system. One set was implemented in a procedural based language, Pascal, and the other in an object-oriented language, C++. Equivalence between the component sets was guaranteed by ensuring that all code met the same fundamental functional and error-handling requirements.

The subjects were divided into four groups as depicted in figure 2.

Half of the subjects implemented the project in Pascal, the other half in C++. Furthermore, a portion of the students from each language were not allowed to reuse at all, while others were encouraged to do so.

The data collected during the experiment measures the productivity of a subject in implementing the target system.

The main variables in the measure of productivity were:

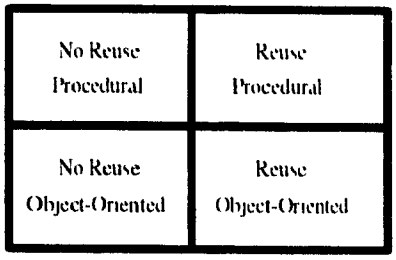


Figure 2: Subject Group BreakDown

Runs - The number of runs made during system development and testing;

RTE - The number of run time errors discovered during the system development and testing;

Time - The time (in minutes) to fix all run time errors.

The secondary variables in the measure of productivity were:

Edits - The number of edits performed during the system development and testing;

Syn - The number of syntax errors made during system development and testing.

Multiple productivity variables were used to provide a complete picture of the development process. It was decided that the **Runs**, **RTE**, and **Time** variables would be given greater emphasis due to their significance in the development process.

Experiment Results

The goal of the experiment was to answer questions with respect to the impact of the object-oriented paradigm versus the procedural paradigm on the successful reuse of software components. We will now address two of these questions (those most relevant to the topic under discussion), separately summarizing the experimental findings as well as giving our own opinions and interpretations of the data.

The tables which follow have a p-value associated with each productivity variable. The p-value is the probability that the difference could have been obtained by chance, rather than reflecting a true difference in productivity. Following conventional criteria, a difference is deemed statistically significant if its p-value is less than 0.05.

1) *Does the object-oriented paradigm promote higher productivity than the procedural paradigm?*

The third column in table 1 lists the means of the productivity variables calculated from all subjects using the procedural language, including subjects who reused and those that did not. The fourth column shows similar means for subjects in the object-oriented categories.

			Means	
	Sig.?	p-value	Procedural	O - O
Runs	Yes	0.0066	59.27	47.50
RTE	Yes	0.0078	65.00	50.20
TIME	Yes	0.0104	354.41	261.70
Edits	No	0.3469	271.55	263.65
Syn	No	0.8675	183.67	202.40

Table 1: Language main effect

The data evaluators at Virginia Tech concluded from the Table 1 data that the subjects using the object-oriented paradigm experienced higher productivity. The values for the three main variables in the O-O column are indeed lower, with p-values well below the 0.05 required for significance. The two secondary variables had surprisingly similar figures for the procedural and object-oriented paradigms.

The data evaluators attributed, due to the nature of edits and syntax errors, the lack of significance of the secondary variables to the subjects' lack of practice using the object-oriented language. This may indeed be a contributing factor, however, we believe that it is only one among many.

Firstly, the very nature of object-oriented programming suggests that it is not as intuitive to use effectively and efficiently as procedural based languages [OBHS86]. For instance, in order to reuse, one must *find*, *understand*, *modify*, and *compose* reusable software parts [JS89]. Reuse in procedural-oriented languages requires less of an effort. The procedure caller supplies a set of actual parameters to conform to the callee protocol. Then, the callee (a server) will provide the client with a specific service routine. However, in object-oriented languages, each service request (message) will be sent to an object which will determine the appropriate service response (method). Unlike procedural languages in which a service request is responded to by a specific

service routine, in object-oriented languages the same service request (method call) could be responded to by different service routines (classes). A callee in a procedural language depends only on the routine name whereas a callee in an object-oriented language depends on both the object (class type) and routine (method) name.

It therefore obviously takes more of an effort to *understand* an object-oriented component as compared to a procedural one. This effort would in our opinion reflect itself in the number of edits and the number of syntax errors (for less experienced coders) as well. Even experienced C++ programmers spend quite a bit of their time *understanding*, and to understand a component, one must edit. We believe that the number of edits would therefore be high, independent of object-oriented experience. You may argue that experienced programmers would require less effort for *understanding* components. We tend to agree, however, this would be offset by a greater amount of effort devoted to efficient class composition and quality object-oriented programming. They would expend greater effort in considering the proper development of concrete and abstract classes, issues such as the ideal number of methods per class, generality of applicability versus payoff and others related to reuse and general object-oriented programming. In this experiment, the details of the object-oriented code quality were apparently (mistakenly) not taken into consideration.

2) *Does the object-oriented paradigm promote higher productivity than the procedural paradigm when programmers reuse?*

The means in table 2 are for both programming paradigms and for subjects who did reuse.

			Means	
	Sig.?	p-value	Procedural All Reuse	O-O All Reuse
Runs	Yes	0.0001	50.07	32.21
RTE	Yes	0.0005	55.71	34.36
TIME	Yes	0.0153	301.86	208.86
Edits	No	0.8380	189.00	208.64
Syn.	No	0.9767	137.14	164.71

Table 2: Reuse (procedural versus object-oriented)

The results in table 2, for the three main productivity variables, suggest a positive answer to the question. Here, too, we have a favorable result for the object-oriented

paradigm. But here, once again, the secondary variables did not differ in the proper direction, and the experimenters failed to rationalize this discrepancy.

Our analysis of the data in table 1 applies here as well. It is important, however, that we rationalize why the number of edits and syntax errors differ even greater in this situation of procedural and object-oriented ALL reuse, table 2.

In table 1 the means reflected subjects who reused as well as those which did not. And the reasons given for the secondary variables' low productivity results were mainly due to software reuse issues. Therefore, one would assume that analysis of data pertaining to **reuse ONLY**, would result in an even greater difference from the hypothesized direction for both Edits and Syntax Errors. This is precisely what we have observed from the data of table 2.

Final Comments

The Virginia Tech experiment has indeed shown that the object-oriented paradigm has a particular affinity for reuse. With regards to their conclusions about productivity, we have a more skeptical view. They claim that, "the object-oriented paradigm substantially improves productivity, although a significant part of the improvement is due to the effect of reuse.". We claim that, "If productivity gains are achieved with the object-oriented paradigm, these gains are directly and mainly attributable to the effect of reuse.".

Notice that our claim begins with the condition "If". It is our contention that productivity gains are achievable with the object-oriented paradigm, however not in its early stages of adoption. Productivity gains are possible, but certainly not guaranteed, once a set of mature reusable classes have been constructed. A mature class hierarchy is one which has been reused fairly extensively and has concurrently gone through several refactoring and testing cycles. Then, and only then, would productivity gains, due to reuse, be noticeable.

During the early stages of reuse-oriented software development, possible productivity gains could be immediately realized in system maintenance activities. This gain would be realizable if and only if the maintainer was also the developer, i.e., the creator/refiner of the application's classes. This way, the effort required in *finding* and *understanding* the affected class hierarchy would already have been expended. In

fact, as Basili points out [Bas90], for effective reuse, it is important to combine the development and maintenance models. Basili even suggests that development should be considered as a subset of maintenance. Upon reflection, this would be especially true in the object-oriented paradigm. The features we have described as encouraging reuse in object-oriented languages, are also useful during maintenance. Modularity makes it easier to understand the effect of changes to a program. Polymorphism reduces the number of procedures, and thus the size of the program that has to be maintained by the maintainer/developer. And lastly, class inheritance permits a new version of a program to be built without affecting the old one, hence *programming-by-difference*. Thus, a set of subclasses actually reflects the history of changes made to the superclass.

Future experiments would be required in order to consider two very important variables, in addition to those studied by Virginia Tech. They are the number of hierarchy levels in the class library and the total number of classes (components). Both have a significant impact on a developer's effort to reuse as described by Woodfield et al. [WES87] who report that small and shallow class libraries ease finding and understanding of classes, hence facilitate reuse and thus increase productivity.

2.2 Object-Oriented Frameworks

An object-oriented framework is a set of classes which provide a foundation for solutions to a set of problems. Frameworks may be domain specific. For example, there are frameworks for VLSI routing algorithms [Gos89] and for drawing editors [VL89, Vli90]. Others are more general and define much of an application's standard user interface, behavior, and operating environment so that an implementor may concentrate on application-specific parts. These general purpose frameworks, often called application frameworks, are the subject of this work. For brevity, we will henceforth refer to them as "frameworks". Examples are: *Interviews* from Stanford University [LCV87]; *ET++* from the University of Zurich [WGM88, WGM89]; *Smalltalk-80* for Smalltalk-80 [KP88] and, *MacApp* on the Macintosh [Sch86].

Frameworks are more than well written class libraries [JF88]. A framework is the design of a set of objects that collaborate to carry out a set of responsibilities, an abstract design. According to Deutsch, the most important part of a framework is

the part that describes how a system is divided into its components [Deu83]. In fact, frameworks can be thought of as application templates for a family of solutions to a set of related problems. A framework (or abstract design) is used by modifying existing classes and/or by extending it via the definition of new subclasses. Whereas class library components are used individually, classes in a framework are reused as a whole to solve a specific instance of a certain problem.

Figure 3 illustrates the design of an application that has been developed with the ET++ framework, an excerpt from the ET++SwapsManager application [EG92]. Bold class names are those of the underlying ET++ framework itself while the classes within the dotted area were added for this particular application.

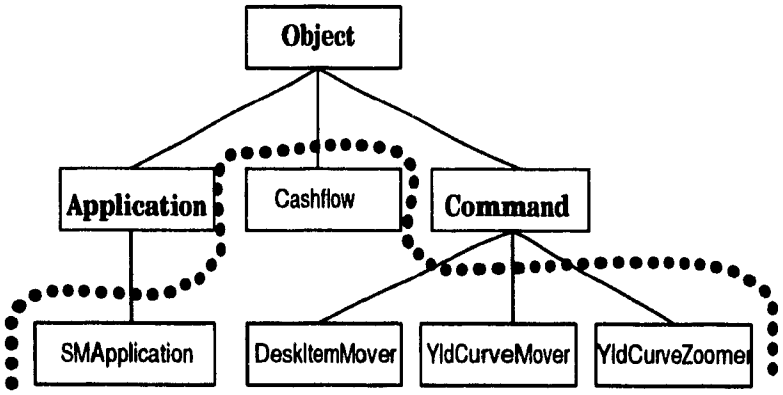


Figure 3: The overall design of an application built on top of a framework

A mature framework will have more than one concrete subclass for a majority of its abstract classes, allowing for most applications to be developed by plugging together existing components. If new subclasses are required, they are usually very simple to code, because mature abstract classes tend to be complete.

The major difference between using object-oriented frameworks and using component libraries is that the user of a component need understand only its external interface. In contrast, the user of a framework must also understand the internal structure of the classes in order to adapt and extend them by inheritance. Frameworks require more training to use and are easier to abuse than component-based frameworks. They do however provide enormous potential for reuse, in fact, reuse is what frameworks are all about.

2.3 Levels of Reuse in Frameworks

Similar to the notion of programming-in-the-small, -in-the-middle and -in-the-large [DK76], we have identified three different levels of reuse in object-oriented frameworks. The first, reuse-in-the-small, involves low-level reuse, i.e. the reuse of a class, method, and/or code fragment. For example, adding a subclass to an abstract class, called specialization, is reusing-in-the-small. Abstract classes are incompletely specified and designed to be subclassed rather than themselves instantiated. An abstract class is actually a small scale design for reuse of the design of small scale components. In contrast, a framework can be considered a large scale design.

The next level of reuse is reuse-in-the-medium. It lies in between the reuse of code and small scale designs of abstract classes and the reuse of large scale designs in frameworks. This is the reuse of objects and their interactions. We shall refer to this type of reuse as "micro-architecture" reuse. As frameworks codify design knowledge of a particular domain, micro-architectures codify design knowledge in terms of the behavior of object collaborations. Micro-architectures reuse both the design and code describing the kinds of objects and the interactions and control flow among them.

Reuse-in-the-medium involves another level of reuse which is a little higher in abstraction than micro-architectures. It involves the interaction of micro-architectures. Therefore, at this level of reuse, the objects are no longer classes, but are themselves micro-architectures. This is in fact the reuse of the architecture of a system which has been instantiated from the object-oriented framework. At this level, for reuse to be successful, the designers must reuse a system which has been instantiated with the same framework with which they intend to do further development and redesign. For example, it would not be worth-while for a designer using the ET++ framework to reuse the architecture from an application developed with the Interviews framework [Ber90].

As a framework describes the architecture of a system which has been instantiated from it, micro-architectures can be considered just that, i.e. micro-architectures of the larger framework architecture.

The highest level of reuse, reuse-in-the-large, is the reuse of objects which are themselves independent systems, systems which are reused as they are, without being modified or extended in any way. We call them "external systems". It is the system

which reuses them, the "target system", which is responsible for adapting itself to the protocol requirements of the external system. The external systems may or may not have been developed with the same framework as the target system, they may even have been developed in a different programming language. The point to be stressed here is that the target system need not bother with the internal implementation and design details of the external systems since there is absolutely no intention of modifying or extending them. Although reuse-in-the-large is not specific nor unique to frameworks, it is an important reuse level that we have had experience and success with in the development of the Macrotec Toolset.

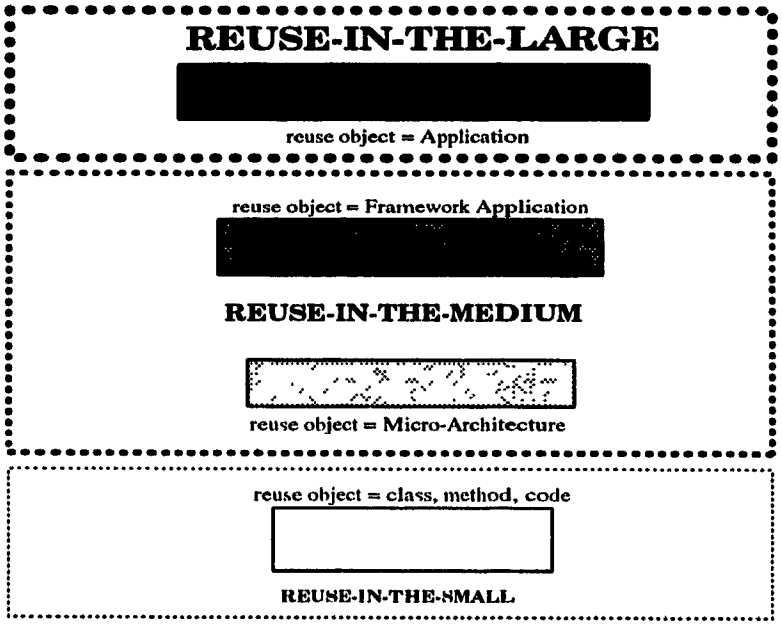


Figure 4: The different levels of reuse and abstraction

Figure 4 depicts the different levels of reuse we have described. Notice the use of shading to represent the level of abstraction. For instance, reuse-in-the-large involves reusing a "black box" object, i.e. an application. Each level will be described in greater detail in later chapters.

Chapter 3

ET++ and the Macrotec Toolset

You will now be proudly introduced to Macrotec, an object-oriented toolset we have developed throughout the past year. The Macrotec development, using the ET++ framework, illustrates and validates many of the concepts described in this work and will be referred to in the later chapters. This chapter concludes with a brief discussion of ET++.

3.1 The *Macrotec* Toolset

We have developed a new methodology for the architectural modelling and high-level requirements specification of business processes and information systems. To support and validate our methodology, we have engineered the *Macrotec* toolset. *Macrotec* currently allows for graphical model specification, automatic graphical layout, logical and performance analysis, and hierarchical decomposition. To support this functionality, various tools were built or integrated into *Macrotec*. Externally, integration is achieved through a seamless user interface, and internally, integration is furthered by one single data representation scheme and a simple yet effective and extensible mechanism for tool interaction. In this section, we present our methodology and focus on the tool development effort that led to *Macrotec*. Specifically, we discuss *Macrotec's* requirements, design and implementation, and we evaluate our development effort.

3.1.1 Introduction to *Macrotec*

In a joint research project, we have developed, in cooperation with DMR Group Inc., a new methodology for business modelling. Our approach combines several concepts that have originally been developed in separate contexts, such as entity-relationship modelling of information, specialization and inheritance in the sense of object-oriented languages, event analysis, and analysis of data (product) flow as well as resource utilization. We have integrated these concepts into a uniform modelling framework with a precise semantics for the dynamic aspects, which has been defined through the formalism of Petri nets.

The resulting modelling approach [BDD+92] supports facilities such as architectural views at different levels of abstraction, and performance analysis, based on the dynamic semantics mentioned above. In many ways, however, it goes beyond current approaches. For instance, it explicitly supports inheritance and specialization, and it includes an expressive set of relationships, yet small enough to make them easy to use. Moreover, our approach lends itself to automation, e.g., semi-automatic substitution of model parts, various consistency checks, and flexible animation (forward and backward).

In order to support and validate our approach, we have developed the *Macrotec*[1] toolset, a tool which will eventually support all facets of our methodology [KLO+93a, KLO+93b].

In the sections which follow, we discuss the requirements for systems supporting our methodology. Then, we detail the design considerations behind Macrotec and provide a scenario of its usage. Next, the implementation of Macrotec is described, together with an evaluation of our development effort. Finally, we present the current status and future research directions of our project.

3.1.2 Toolset Requirements and Existing Tools

In this section, we describe the functional requirements of the Macrotec toolset. We then report on our evaluation of existing tools against these requirements and detail the tools we have integrated into Macrotec.

[1] "Macrotec" is a contraction of the words *Macro*scope, our project's name, and archi*tec*ture.

The toolset must include:

- A tool for the graphical editing and validation of models (i.e., their representation as annotated graphs or networks), complemented with facilities for automatic graph layout;

- A dynamic analysis tool supporting both timed animation and performance analysis. The animation engine (we use the terms "engine" and "tool" interchangeably) must graphically and interactively simulate the execution of the model, thus enabling visualization of the model components' interactions. Furthermore, it must support forward and backward execution in order to answer questions such as: What are the actions that consume given inputs and in what order do they occur? What are the outputs obtained? What are the necessary intermediate actions to be performed? What inputs are required for a specified output?

  The performance analysis engine must analytically generate quantitative results such as action throughput, bottlenecks, resource utilization and waiting time for actions. This way, potential problems (e.g., loops) may be revealed, and the impact of modifications on the system design may be more easily understood;

- A substitution tool supporting multi-level modelling. This tool must provide mechanisms for the decomposition and abstraction of network parts into subnets and super-nets, describing, respectively, lower and higher levels of abstraction. Obviously, these mechanisms should preserve visual and behavioral consistency between different-level nets in respect to their adjacent nodes;

- Facilities for data exchange and evolutionary design, namely, a state-of-the-art database and a standard data exchange format.

Given this extensive list of requirements, we strove for using existing tools and conducted an evaluation. In what follows, we describe our principal findings.

Most existing modelling tools are based on one of the following models: data-flow, entity-relationship, object-oriented or Petri net model. Our methodology does not exclusively support these underlying models but rather merges their main concepts into one coherent approach.

The dynamic analysis and substitution tools we evaluated, including *DesignCPN*, *Eval*, *GSPN*, *MetaDesign*, *RDD100*, *SPNP*, *Voltaire* [KOR92] and Franck's system [Fra92] did not meet the above requirements, since they provided insufficient support for one or many of the following criteria: timed dynamic analysis, backward animation, automatic performance analysis results generation, integrated view of static and dynamic modelling, concurrency and parallelism, data exchange and model evolution, hierarchical models, and multi-level validation.

These tools offer, at best, partial solutions to our requirements. Thus, we decided to develop an integrated toolset based on existing tools that fully satisfy one functional requirement, and on tools built from scratch. Figure 1 depicts the different tools involved in the resulting Macrotec toolset. We have integrated the *SPNP* performance analysis tool [TMWH92] and an automatic graphic layout package being developed at the University of Toronto [MEN92]. The modelling, animation and substitution tools [JBB+92], however, were all developed by our group.

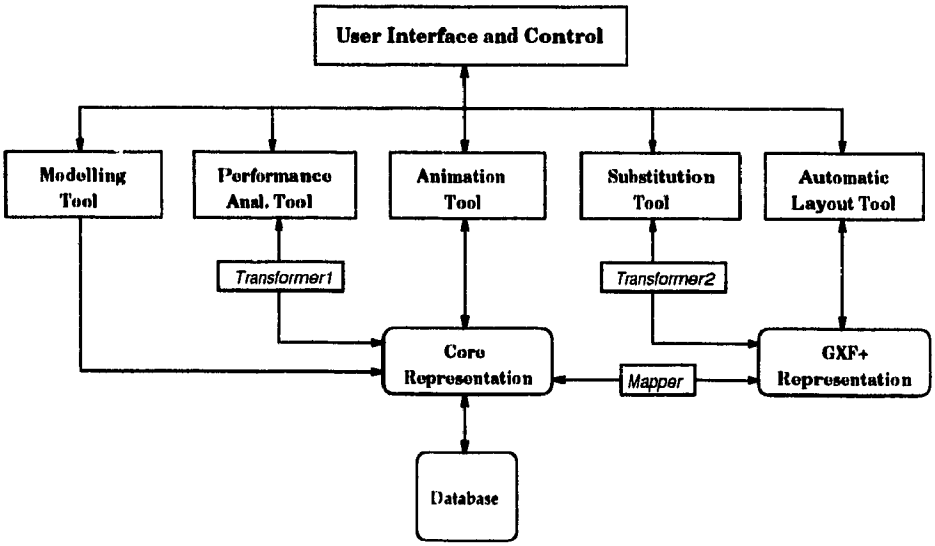


Figure 1: Macrotec Architecture Overview

3.1.3 Design of the Macrotec Toolset

The design of the Macrotec toolset was driven by three major considerations: internal and external integration, and extensibility. We felt that these guiding principles

would be essential to our design, if Macrotec was to support all of the functionality mentioned in the previous section, and possibly more in the future.

Internally, the *core representation* is the heart of Macrotec (see figure 1). All information to and from the user interface is, after manipulation by the various tools, managed in the core representation. Internal integration in Macrotec is furthered by the underlying storage facility, the Gemstone system[2], an object-oriented database management system allowing the storage and retrieval of the core representation, and supporting simple versioning.

Macrotec consists of two categories of tools. In the first category are the tools that manipulate graph layout data. Such tools store their data in the *GXF+ representation*. GXF+ [KLS93] is our customized version of *GXF*, a standardized graph exchange format [MEN92]. Supporting GXF+/GXF allows us to easily exchange data with other, special-purpose, GXF-based systems such as the automatic layout tools being developed at the University of Toronto. Non-GXF+-based systems require data transformation programs. For instance, integrating our substitution tool (implemented before adopting the GXF+ standard) required the development of the *Transformer2* program. Mapping of the core into the GXF+ representation and vice versa is carried out by the *Mapper* component.

Tools belonging to the second category manipulate the model data that are not related to the graph representation. In case these tools are part of the Macrotec process, e.g., the animation tool, they interact with the core directly. Otherwise, a data transfer program to and from the core is required. For instance, the performance analysis tool, running as a separate process, interacts with the main Macrotec process via *Transformer1*.

By external integration we mean that the user's interactions with all the tools and facilities of Macrotec are as uniform and comfortable as possible. This is achieved by having the user interact with the system via one single base window, giving him or her access to the complete functionality of the system and allowing for easy switching between the different tools. User interface prototyping and software reuse at the design level were instrumental in accomplishing this type of integration.

Extensibility in Macrotec is furthered by a loosely coupled yet efficient architecture. Since the different tools, while running in parallel, do not interact with each

[2]Gemstone is a registered trademark of Servio Corporation.

other, Macrotec's control component can be kept quite simple. It synchronizes tools, transmits external events to the tools, and controls access to the core representation, using the inter-process communication mechanisms provided by Unix.

3.1.4 Using Macrotec

One typical application domain of Macrotec is the modelling of enterprise information systems for prescriptive usage. In this section, we provide a scenario of this kind of modelling. We model, as an example, a delivery system with products being ordered, transported and finally delivered to their destination.

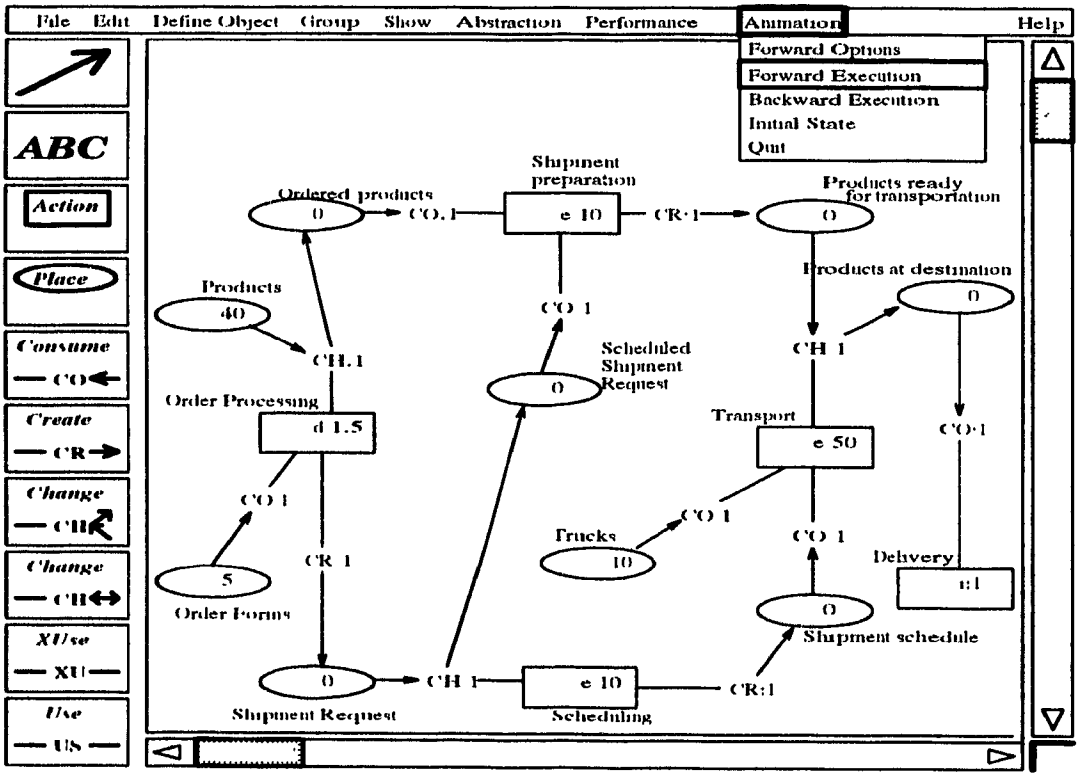


Figure 2: Sample model *Delivery System* with animation and performance analysis attributes

Figure 2 shows the base window through which the user has access to the full functionality of Macrotec. The window consists of three distinct areas: the menu bar which gives access to the animation, performance analysis, substitution and automatic

layout tools; the palette in which the user may select an icon for editing actions, places, relations and their attribute values (our models' building blocks); and the drawing area in which the user may display and edit the network. In figure 2, the user has loaded a network, possibly reusing a template or parts of an existing model. The user may edit and refine the network, adopting a top-down or bottom-up approach by respectively decomposing or abstracting parts of the network. The built-in validation component of the modelling tool makes sure that the different levels of the network are consistent.

The animation engine is triggered through the base window via the pull-down menu shown in figure 2. The graphical execution of the model may be performed at user-specified hierarchical levels, allowing for partial model assessments and permitting the user to define a configuration that corresponds to his or her mental model of the system.

Performance analysis generates quantitative results on model behavior. For example, a comparatively low action throughput and average number of entities in an action's input place ("low" might have a different significance depending on the network architecture) may confirm a bottleneck that has already become apparent during animation. In light of these results, the user may increase the number of over-strained resources to smoothen execution (in our example, we could increase the number of "Trucks", if the action "Transport" had low throughput). Similar to methodologies and tools in related domains [SGME92], the user may prototype the model. He or she might run, in an iterative way, animation and performance analysis, until the appropriate attribute values and configuration for a desired model behavior are found.

3.1.5 Implementation and Evaluation of Development Effort

Macrotec is a Unix-based system developed mostly in C++ (minor parts were written in C), running under SunWindows, NeWS, and the X11 window system. Its user interface has been implemented with the *ET++* application framework [WGM89] and as database we are using Gemstone. To date, we have invested more than four person years in its development.

The data transformation programs required to integrate external tools into Macrotec are an indicator for the extensibility of the system. According to our experience, those programs dealing with the GXF+ representation tend to be considerably longer (4:1 ratio in lines of code) and more complicated than the ones interacting with the core representation. However, this will not be a severe limitation, since future Macrotec extensions will most likely require a transformation of type Transformer1. We do not see the need for further graph manipulation tools, and hence transformations of type Transformer2 will not be required.

We have adopted an object-oriented development approach that has provided significant productivity and quality gains. For instance, our user interface software is highly flexible and reusable, in part due to the use of ET++. ET++ is a powerful, object-oriented class library integrating user interface building blocks with high-level application framework components. The usually steep learning curve for such application frameworks has been alleviated by the use of some powerful C++ development tools, most notably, the *Sniff* tool[Bis92]. Another benefit of the object-oriented approach in Macrotec is the use of Gemstone together with its C++ interface which has led to an efficient database interface.

To prototype the user interface of Macrotec, we ran several user interface development cycles, using tools such as *HyperCard* and *Chiron-1* [KCTT91]. The integration of several standalone tools with their own graphical user interface was an incentive to meet (or surpass) the usability criteria of each.

As our system evolves, with new external tools requiring integration, we will be in a better position to determine how easily our system can be adapted and hence whether or not our design and guiding principles are indeed sound.

3.1.6 Current Status and Future Work

A prototype version of the Macrotec toolset has recently been completed. Preliminary experience indicates that the prototype efficiently and effectively supports our methodology. In the current version, backward animation and model substitution are not yet fully supported. We intend to further validate our methodology and toolset by applying it to large examples and by using it with evolving systems. Further plans include customizability at the presentation level, the addition of a model documentation and elicitation component, and navigation aids for substitution hierarchies.

3.2 The ET++ Framework

We will now describe in greater detail the ET++ framework and relevant design details. We will then compare Macrotec with *Draw*, an application developed using ET++ and reused heavily in our development effort.

3.2.1 ET++ and Relevant Design Issues

ET++ was developed at the University of Zurich by Gamma, Marty and Weinand. ET++ is a class library of C++ classes, approximately 250 of them, which aims to provide facilities comparable to the ones found in the standard Smalltalk class library. The classes are distributed amongst 9 hierarchy levels as depicted in figure 3.

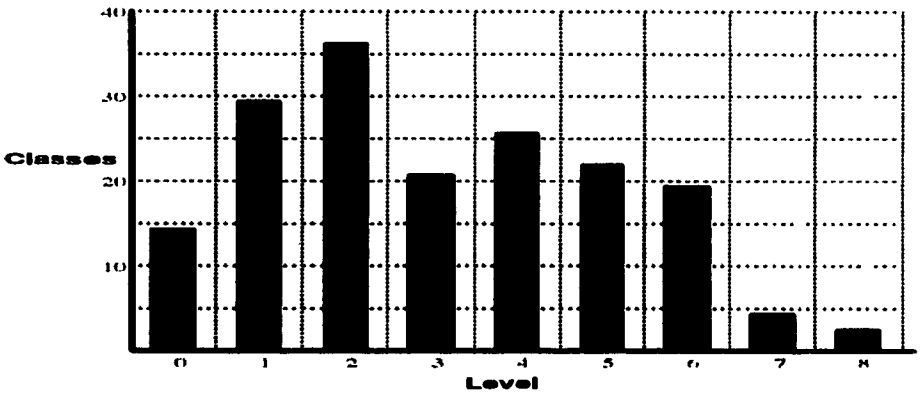


Figure 3: Distribution of Classes in the ET++ Hierarchy

ET++ is structured as a single-rooted inheritance hierarchy with many virtual functions available to provide flexibility and opportunities for extension. The backbone of the ET++ architecture is a small device-dependent layer mainly mapping an abstract window and operating system interface to an underlying real system (Figure 4).

The *Basic Building Blocks* contain the most important abstract classes of the ET++ class hierarchy. All classes in ET++ inherit from class *Object*, which defines the protocol for actions common to all classes. Most of the member functions in Object do nothing, and exist purely to be over-ridden in subclasses. Every object that appears on the screen is a subclass of VObject (visual object). VObject defines

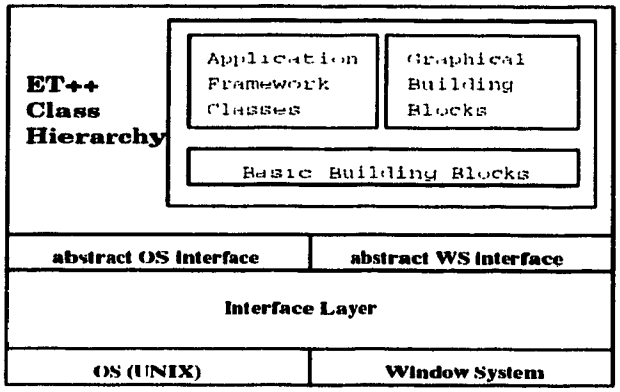


Figure 4: ET++ Architecture Overview

an abstract protocol for the behavior common to visual objects - managing their size and position, drawing them, handling input, etc. As the root of the hierarchy of visual objects, it is analogous to Object in the wider system. Basic building blocks are also supported, like arrays, lists, sets etc.

Application framework classes are high-level abstract classes that factor out the common control structure of applications running in a graphic environment and thus provide support for the overall functionality of interactive applications. The classes *Application*, *Document*, *View*, and *Command* are responsible for the overall application behavior. They define the abstract model of a typical ET++ application and together form a generic ET++ application *Application* is in overall control, and manages any number of *Documents*. *Document* holds the data structure or model of the application, and is responsible for loading and storing this data in files. All modifications to a *Document* are implemented as Commands, and the *Document* permits the user to undo the last command performed. The class *View*, another subclass of *VObject*, represents an abstract and possibly arbitrary large drawing surface. Its main purpose is to factor out all control flow necessary to manage rendering and printing as well as maintaining the current selection. A *Document* can have any number of *Views*.

The *Graphic Building Blocks* contain all the graphical and interactive components found in almost every user interface toolbox, such as menus, dialogs, or scrollbars. In addition, it defines the framework to easily build new components from existing ones.

The System Interface Layer provides its own hierarchy of abstract classes for operating system services, window management, input handling, and drawing on various

devices. Subclasses exist for implementing the system interface layer's functionality for various window systems and the UNIX operating system.

3.2.2 ET++ Run-time Information about Classes

One problem of C++ is that its run-time system does not provide any information about the class structure or the instance variables of an object. It can be very useful in object-oriented systems to be able to ask an object what class it is, or to know what instance variables it possesses. ET++ therefore implements a class *Class* which is itself a subclass of Object and is analogous to Smalltalk's metaclass. The basic idea is to call a macro in the definition and implementation part of a class. The following example shows a class conforming to this ET++ coding convention.

```
//file Example.h

class Example:public Object {
class Collection *col;
int size;
char *name;
public:
MetaDef(Example)
Example();
//...
};


//file Example.C

MetaImpl0(Example);

Example::Example()
{

   //....
```

```
}
```

In the simplest form both macros, just take the name of the class as argument. Information about the instance variables can be specified with the **MetaImpl** macro instead of the **MetaImpl0**. In this macro, the instance variable's type and name are enumerated:

```
MetaImpl(Example,(I_O(col), I_I(size), I_CS(name)));
```

The symbols I_? are used to specify the type of an instance variable. This macro version is needed in order to give support to the ET++ programming environment. The only time we made use of the programming environment was to determine the class type of a particular graphical object. The use of **MetaImpl0** is therefore sufficient.

Not conforming to this macro convention results in not being able to test the dynamic type of an Object with the IsKindOf method. We learned this the hard way and it cost us at least half a day of debugging before finding the problem and finally tracing it back to this.

3.2.3 ET++ *Draw* application versus *Macrotec*

ET++ classes are not documented nor are there any manual pages. Therefore the best way to learn ET++ is to study example applications. We studied Draw (a full fledged drawing editor) and soon realized that it supported many of our requirements for Macrotec. We therefore decided to make use of as much of the original Draw application's code as possible. We modified many of Draw's classes by modifying existing methods, adding new methods and removing others. We also created several of our own classes and integrated them into the existing Draw hierarchy. The original Draw application required close to 4000 lines of code and Macrotec is now at approximately 10,100 lines.

Chapter 4

Design and its Reuse in Frameworks

This chapter describes a major contribution of our work, i.e., the use and reuse of frameworks. We begin with a discussion of why design has long been considered a difficult task. Then, the two lowest levels of reuse, reuse-in-the-small and reuse-in-the-medium, are described with a concentration on the latter, specifically the reuse of micro-architectures.

4.1 The Challenges of Design

Design has long been considered hard and has been at the heart of most difficulties in software systems. No matter how carefully prepared a system specification might be, in the end, many easily specified tasks are "easier said then done". Software design is difficult in part because designers, in most domains, work in the unknown, unguided by the history and traditions of earlier designs. Only a few domains have been worked to the point that their design principles are well understood. Those domains which immediately come to mind are compiler and operating system design. Designing within these domains is simplified by the tremendous amount of design experience from which one can confidently be guided. We cannot, unfortunately, easily and safely map design principles from these few "design wise" domains to the many others. We are, in other words, missing basic pivot points around which further

design decisions may be safely and easily based. This means that currently, design is indeed "experience limited".

Another general problem is the effect of changing design parameters. The designer of an integrated circuit can easily predict the effect of an increase in voltage on the individual components and therefore on the overall functionality of the circuit board. The effect of changing software design parameters are much more difficult, if not impossible, to predict. When dealing with frameworks, for instance, design parameters include: number of classes, class protocols, class size, inter-object protocols, etc. Changes in any of these parameters could affect functionality, reliability, generalizability, reusability, etc. of the underlying framework(s).

The design of flexible and reusable software is even harder. In the traditional waterfall software lifecycle model, design is performed once and for all, after initial system definition and requirements are complete. However, we are asking designers to deal with changes in system requirements throughout the life time of the software product. Additionally, the ultimate goal of "design for reuse" is reusable software, which requires the design of general, extensible software components. Design for a fixed, specific set of requirements is difficult enough, and yet we are now asking designers to predict the future uses of software and to incorporate the requirements for these possible future applications into the current design.

Finally, to complicate things even more, a system should be designed to support non-functional requirements. These include reliability, robustness, performance and many others which are difficult to control due to the many unexpected uses/input of a system.

We find it appropriate to end this section with a quote from Brooks, "We can get good designs by following good design practices instead of poor ones. Good design practice can be taught. Whereas the difference between poor designs and good ones may lie in the soundness of the design method, the difference between good designs and great ones surely does not. Great designs come from great designers." [Bro87].

4.1.1 Frameworks to the Rescue

Thankfully, there are frameworks, an object-oriented technique to facilitate design level reuse.

In order to alleviate the problem of the designer working in the unknown, we need a means to limit the huge number of design decisions a designer must take. Most designers have recognized that working from a set of fixed low-level components (abstract classes and subframeworks) provides many advantages to the design exercise. This is due in part to the fact that design is an iterative process in which design decisions may be guided by these low-level reference points and by the higher level, more abstract design, of the underlying framework. This approach can certainly not be considered as "top-down" design. However, we would not immediately tag it as "bottom-up" either. In fact, design in the presence of reusable design abstractions will be somewhere in between.

During a panel on "Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks" at the OOPSLA 1990 conference in Ottawa [WBVC+90], object-oriented gurus stressed the bottom-up design approach in the development of frameworks. Here, we are however not involved in the development of a framework (heaven forbid), we are dealing with issues in the design of applications based on an already developed framework. We therefore suggest that we "design with guidance" from the framework's design and possibly the design of subframeworks and that we take a mixed top-down/bottom-up approach.

In practice, we have found that the framework which underlies a complex system such as *Macrotec*, actually guides the design of subframeworks for subsystems. A major characteristic of frameworks is that they are designed to be refined into subframeworks which can be applied to specific subsystems. These subframeworks may themselves guide the design of other subframeworks and so forth. This stepwise refinement of frameworks thus allows the designer to cope more easily with the constraint of changing requirements and software extendibility. Section 4.2 will be devoted to this idea of "framework refactorings".

Using frameworks thus allows designers to concentrate on the application-specific design issues. The designer is provided with both top-down and bottom-up support in the design decision process. Thus, in designing applications based on frameworks, design decisions unavoidably result in the modification or introduction of subsystem frameworks or even in changes to the underlying, supposedly stable framework. These design decisions will result in eventual changes to the configuration of framework components and, possibly, to the creation of new components, i.e. new subclasses of existing classes).

As systems grow in complexity, the architectural dependencies between its component parts may become dangerously dense. There are different techniques which can be applied to maintain or even improve the quality of the original framework-based design, even after having applied architectural transformations. These techniques will also apply to coping with the changes made directly to a framework's design parameters (as mentioned in the previous section). Section 4.2.1 will be devoted to describing these techniques.

The framework-based design reuse technique mentioned above is only now beginning to receive greater attention for its potential for reuse pay off [WBJ90]. Sections 4.3 and 4.4 are dedicated to this topic.

The remaining issues, such as safety and reliability, are, to a large extent, automatically dealt with by the very nature of frameworks. Components in a framework are, or at least have the potential to be, more reliable, since they should have been thoroughly tested and proven in previous applications.

4.1.2 ET++ Design Reuse in *Macrotec*

A framework, such as ET++, hides the parts of the design that are common to all of its possible instances. In what follows, we mention two design reuse examples we experienced when working with ET++.

ET++ represents the design (architecture) of an application in the same way an abstract class represents the design for subclasses. Therefore, using the framework will facilitate design considerably, freeing the application designer from common, low-level and often mundane design issues. Also, the ET++ framework is valuable because it allows the reuse of design parts which are often difficult to understand.

For instance, in developing Macrotec we did not have to consider low-level system issues. ET++ has, as system interface components, a hierarchy of abstract classes. These classes encapsulate operating system services, window management, input handling, and drawing on various devices. The two abstract classes **SYSTEM** and **WINDOWSYSTEM** define the entry point into the system interface hierarchy (cf. Chapter 3 figure 4).

Another example of reusable design in ET++ is the mechanism called *change propagation*. This mechanism allows graphical objects to synchronize their states,

e.g., change their presentation, according to changes in observed objects. The class OBJECT defines the subframework to synchronize the state of different objects. There is a method to register an object as dependent on another, AddObserver(). Modifications to the state of an object are announced with a Changed() method, triggering a call of the DoObserve() method for all dependent objects. To react to a change notification, this DoObserve() method has to be overridden in the participating graphical objects. In Macrotec, we use change propagation to maintain connections between graphical elements and to ensure that, when their positions are changed via "click and drag" operations, all their dependent graphical objects are updated.

In conclusion, ask yourself the following question: Have I ever worked with power designers? If so, they probably told you that many designs occur again and again and made comments such as, "Oh, that structure is very similar to a widget and, typically, you handle widgets this way [BR87]". Rarely will they begin from scratch. Although in this context, the reuse potential is directly related to the designers experience, the fact remains, according to Biggerstaff [BR87], that design reuse is the only way we can even come close to an order of magnitude increase in productivity or quality.

4.2 Reuse-in-the-small

For reuse-in-the-small in object-oriented frameworks, we distinguish two approaches, extension-by-addition and extension-by-modification. Extension-by-addition involves framework restructuring techniques. Extension-by-modification occurs when an object-oriented system is built by reusing an existing system. In this section, we will describe these approaches and illustrate them with examples from the Macrotec development.

4.2.1 Extension-by-Addition - Framework Refactoring

The main reason that framework design iterates is because frameworks are supposed to be reusable. According to Garlan [Gar83], it is not possible to reuse software until it is written and working, and therefore, iteration is a must.

Iteratively refining the framework, subframeworks and abstract classes (recall an abstract class is a small scale design, i.e. a template for concrete classes) is the normal

procedure of "maturing" an application's framework[1]. These iterative refinements usually require a lot of hard work involving structural changes to the framework hierarchy, a process we call refactoring. To motivate the practical importance of refactoring in designing framework-based applications, we will present examples we came across during the development of *Macrotec*.

Refactoring To Generalize: Creating an Abstract Superclass

Refactoring to generalize can be seen as an example of iterative bottom-up design as mentioned in the previous sections. As the design of an application's framework matures, general concepts can usually be derived from specific examples. The specific examples are implemented in concrete classes. As common abstractions are determined, it is often necessary to separate this common behavior and to move it to a new abstract superclass for the set of concrete classes [SD91].

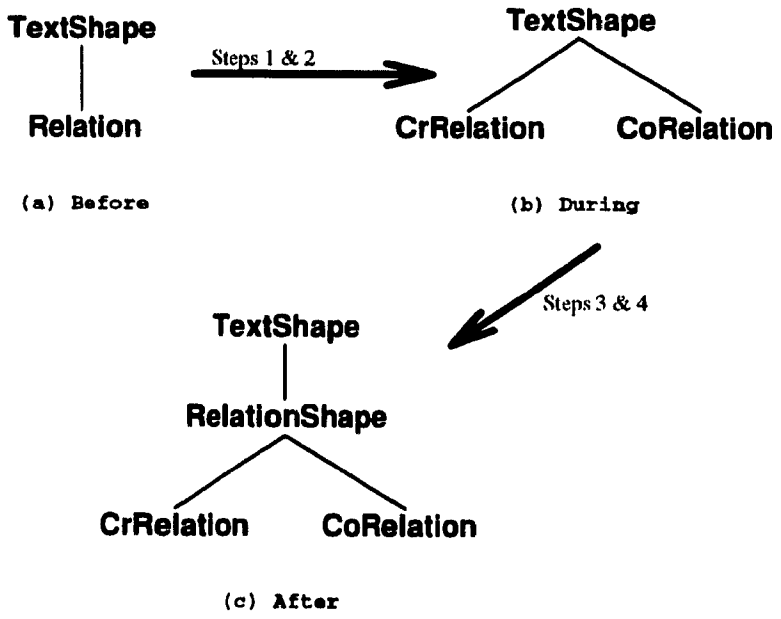


Figure 1: Macrotec class hierarchy - Creating an Abstract Superclass

During the early stages of the *Macrotec* development, the class hierarchy, with respect to the relations between *actions* and *places*, was as depicted in figure 1(a).

[1] *application's framework* (the architecture (class hierarchy) of an application, e.g. Draw, developed from an underlying application framework e.g. ET++) should not be confused with *application framework* (the underlying framework, e.g. ET++, with which one develops applications).

TEXTSHAPE is responsible for drawing text onto the drawing area (setting display properties, making the drawn text dependent on other objects, etc.). RELATION includes methods to maintain which shapes (OvalShape, for places and BoxShape, for actions) it is connecting as well as database save methods. An early stage of development had *Macrotec* supporting only one relation, the "create" relation. After having fully implemented and tested the create relation, we began introducing the other relations. We soon realized, however, that most of the methods and data members were the same for all the relations. A series of changes, shown in figure 1, were made to generalize the Relation class in order to support both CrRelation and CoRelation relations:

1. the Relation class was renamed CrRelation,
2. The CoRelation class was added as a subclass of TextShape; data members and methods were copied from the CrRelation class, and modified,
3. the Relation class was added as a subclass of TextShape and as an abstract superclass of CrRelation and CoRelation,
4. the data members and methods common to CrRelation and CoRelation were migrated up to Relation, their common superclass.

This was a fairly straight-forward procedure since all the methods which were migrated to the superclass had exactly the same implementation, in either relation. If this had not been the case, we would have had to introduce new methods. In general, if class A and B have a slight difference in their implementations for method Y, the differences have to be separated out. This requires that new methods be defined in each subclass, A and B, in order to take care of the differences. The differing code in Y must then be replaced by calls to the new methods in each of the subclasses. Then and only then can the method Y be moved up to the abstract superclass.

Refactoring To Specialize: Subclassing

Generalizing using abstract classes is usually accompanied by specializing using subclasses. Common abstractions are captured in abstract superclasses, whereas case-specific behavior is handled in the subclasses. During the development of *Macrotec*,

we specialized several abstract superclasses. Here we show the specialization of the class SHAPESKETCHER with the class BOXSKETCHER.

The ShapeSketcher class is responsible for displaying to the user a graphical shape's bounding box as it is being dragged and stretched on the drawing surface. It also saves the resulting shape in a list of shapes list. We, however, wanted the shapes for our actions (and places) to be of a fixed size, i.e. we wanted the user to select the Action shape from the palette and have the system place the shape at the first location clicked in the viewing area. To do so, the following steps were taken:

1. an empty class, BoxSketcher, was created as a subclass of ShapeSketcher;
2. ShapeSketcher was studied and it was determined that BoxSketcher needed to override the methods **TrackFeedback()** and **SaveDoit()**;
3. the constructor of BoxShape was determined to have to initialize the base class (ShapeSketcher) through a member initialization list.

Quite often class methods have conditional statements that each test for the same set of conditions. When observations of this type are made, this may suggest that subclasses should be defined corresponding to those conditions. In developing *Macrotec*, we did make such observations. For instance, a method in the RelationShape class, called **DBSAVEConnect()**, had the following conditional code:

```
if (this->IsKindOf(CoRelation))  re_name=strsave("Consume");
if (this->IsKindOf(ChRelation))  re_name=strsave("Change");
if (this->IsKindOf(Ch2Relation)) re_name=strsave("Change");
if (this->IsKindOf(CrRelation))  re_name=strsave("Create");
if (this->IsKindOf(UsRelation))  re_name=strsave("Use");
if (this->IsKindOf(XuRelation))  re_name=strsave("XUse");

if  (this->IsKindOf(ChRelation))
    {
     DEBUGP(printf("It's the Ch relation!!\n"));
     anRelation = new MA_RelationAPP(a,p,p);
     }
```

```
else
    if(this->IsKindOf(Ch2Relation))

      ...etc...
```

Since there were already separate concrete classes for the relations create, consume, change, use and Xuse, we simply added a virtual method in the RelationShape class with each relation implementing its own DBSAVEConnect method.

Summary

The previous examples were quite straight-forward. In general, however, structural changes to a class hierarchy may be fairly complex. In particular, changes made lower into the hierarchy, i.e. closer to the root, will require a lot of attention in order to ensure that the restructuring is behavioral preserving [GTC+90].

Inadequate inheritance structure, missing abstractions in the hierarchy, overly specialized classes, may seriously reduce the reusability of a framework. It is important for a framework to evolve in order to eliminate such problems and hence improve its reusability. Successful refactoring will result in cleaner and more comprehensible design, in less code and therefore possibly in faster execution speed.

4.2.2 Extension-by-Modification - Editing Existing Classes

The second facet of reuse-in-the-small, extension-by-modification, typically occurs in a reuse in the medium context. We recall that reuse-in-the-medium involves the reuse of micro-architectures and interactions thereof. This is in fact the reuse of the architecture of an application which has been instantiated from the framework. The reuse of this application implies reuse-in-the-small, both as, extension-by-addition and as extension-by-modification.

We agree with Lieberman [Coo87] that one wants a small extension in behavior to require just a small extension to code. He further contends that adding new code is good, whereas modifying existing code is bad. We share this opinion as long as the code is the code of the underlying framework itself. However, we claim that modifying

existing code, extension-by-modification, is good and desirable, if the extensions are to the application having been developed with the underlying framework (and not to the framework itself).

In other words, extension-by-modification should only be exercised on the architecture components of an application developed from the underlying framework. Although extension-by-modification reuse runs a higher risk of causing reuse upset[2], its payoff potential, as we have experienced in reusing DRAW, is worth the risk.

One of our experiences with this level of reuse was the implementation of the shape with which an Action in Macrotec is represented on the screen. An Action shape is a rectangle of a default size that is positioned at the first location clicked in the drawing area.

Our starting point was the original class hierarchy of **Shape** graphical objects as provided by the DRAW application (see figure 2).

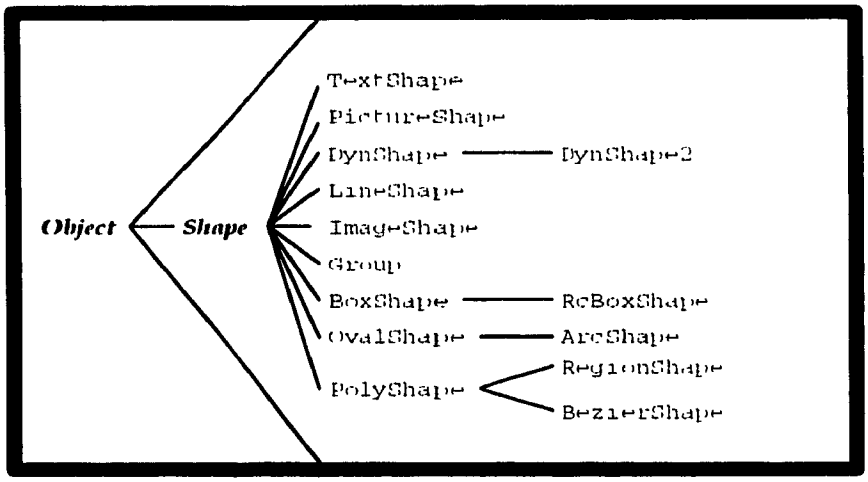


Figure 2: DRAW class hierarchy - Shape graphical objects, **Object** and **Shape** abstract classes of the underlying framework and application's framework respectively.

We considered and discarded the following reuse options for the implementation of our Action shape class:

- **Subclassing BoxShape.** We did not, however, because concluded that BoxShape

[2]Reuse Upset: A term we have introduced, meaning the advantages claimed by reuse are lost and actually worse than if there had been no reuse at all.

was not a valid generalization. It was in fact a concrete class with instantiations in the original DRAW application. However, we had originally agreed to subclass only abstract classes, since abstract classes do not have to provide data representations and future subclasses can use any representation without fear of conflicting with the ones that they inherited [JF88];

- **Creating a new class, subclassing class Shape.** We did not, however, because the functionality of BoxShape, a subclass of Shape, was close to what we required for the Action shape class. The only difference was in the drawing style used and the size of the resulting shape.

Our solution was to remove remove the class RcBoxShape and to reuse the class BoxShape by directly modifying its code, extension-by-modification. This type of reuse involves principles very similar to those of reuse-by-addition. We are in fact introducing a new subclass, however, some of its implementation is not our own, i.e. we are reusing already coded data members and methods. Unlike the reuse of abstract classes by subclassing, in which one may assume[3] correct code, concrete classes have not necessarily been thoroughly tested. The reused concrete class will therefore require testing in the same way a class added via subclassing would require testing. There is however a difference. A class (abstract or concrete) from the application's framework has code "not invented here". Therefore, if ever testing detected a problem, we would have to debug another programmer's code. Obviously, this could result in time consuming debugging sessions and, possibly, in reuse upset. One should therefore carefully consider the following factors when reusing an application's code:

Reliability - the code to be reused should be fairly reliable. A good indication of reliability is the successful use of the application in the past;

Code Changes - the code to be reused should require the least amount of changes, due to system enhancements and/or maintenance. This is however difficult to control and predict. Therefore one should consider the following additional factors which directly affect the ease with which code may be enhanced and/or maintained:

[3]This assumption is based on the current practice of developing application frameworks in which a substantial amount of time is invested in creating and testing abstractions.

Code Complexity - you should not take the chance of reusing complicated code. If ever it required changes, the initial gains experienced due to reuse would quickly be lost and reversed, i.e. reuse upset;

Code Size - one should keep in mind that the greater the number of code lines in a method, the greater the probability of an error within that method.

Extension-by-modification of an application's abstract classes, e.g. class Shape, raises the same concerns as those mentioned above when directly reusing code. However, when extending abstract classes, we should be even more careful since the potential for expensive errors is increased.

We, for example, removed some data members and introduced new ones to the abstract class **Shape**. Such changes, to be behavior preserving, must satisfy certain preconditions. Two simple examples are the following:

A. Remove Data Members, i.e. delete variables at the class level.

Precondition: $\Rightarrow$ *the variables being deleted must be unreferenced.*

B. Add New Data Members, i.e. add an unreferenced locally defined member variable to a class.

Precondition: $\Rightarrow$ *the new data member name must not clash with an existing data member or global variable.*

Opdyke [OJ90] has identified 26 preconditions for such low-level refactorings. They support the high-level refactorings of section 4.2.1 and are naturally applicable to all other changes to a framework class hierarchy as well. After having studied them, we found ourselves verifying the associated preconditions each time a change, any change at all, was required in either an abstract *or* concrete class.

Summary

The reuse of BoxShape and Shape was a big payoff for us in terms of implementation time and code quality (those who originally developed DRAW were seasoned programmers). In fact, in reusing an application, many unknowns, in terms of implementation, may be resolved by studying the functionality of the existing code.

For instance, after having performed the above mentioned refactorings, we had the class hierarchy properly reflecting the class BoxShape as a subclass of class Shape and without subclasses. We were however at a loss as to how we could change the behavior in order to draw a rectangle with default size at the first location clicked in the drawing area, i.e. eliminate the rectangle sketching. When running the original DRAW, we realized that the shapes PolyShape, RegionShape, and BezierShape did not have the same drawing functionality as the others. We therefore decided that, in order to properly change the behavior of BoxShape, (i.e. make the changes in the correct methods and/or classes) we would have to determine how these three shapes had implemented their unique behavior. The first logical step was to compare their class definitions. We immediately realized that each of these three shapes had their respective sketcher classes defined in their .h files. We inspected the class hierarchy and indeed we found shape sketcher classes (see figure 3).

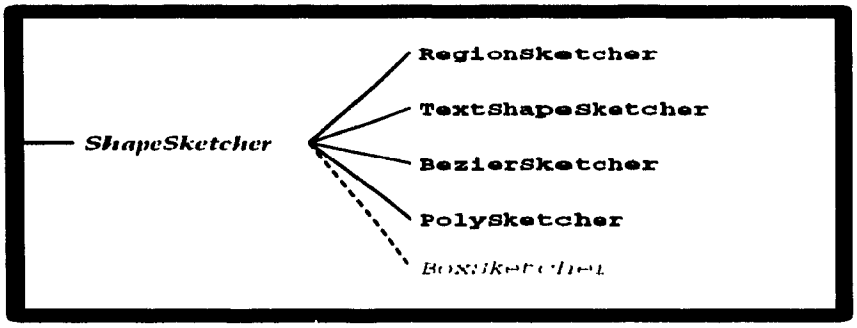


Figure 3: DRAW class hierarchy - Shape Sketcher classes

We now realized that, to change the drawing behavior of BoxShape, we had to introduce a new class, BoxSketcher. We did, and using RegionSketcher as a template (with few changes required to the code) we succeeded in the implementation of the required behavior.

4.3 Reuse-in-the-medium - Framework Applications

Before describing the lowest level of reuse-in-the-medium, i.e. the reuse of micro-architectures, we will discuss in this section the highest-level of reuse-in-the-medium, i.e. the reuse of an application developed from an underlying framework. We feel

that this order of presentation will facilitate the readers effort to properly understand our detailed explanations of micro-architecture reuse.

We experienced this higher-level reuse-in-the-medium when developing *Macrotec* from *Draw*, a preexisting application developed with the ET++ framework. *Draw* is a general-purpose graphical editor similar to MacDraw, allowing the editing of lines, graphical shapes, text, spline curves, etc. When studying *Draw*, we realized it matched very closely the functionality required in Macrotec's graphical modelling component.

In order to successfully accomplish framework application reuse, it is essential to be able to differentiate the underlying framework classes from the application-specific classes. This is important for many reasons, including,

- Much can be learned from application class implementation in terms of the interaction with the underlying framework, for instance,
  - the classes of the underlying framework which were subclassed
  - the methods which were overridden when subclassing
  - the general programming style adopted by the application developers, i.e. naming standards, approximate number of code lines per method, etc.
- If the chosen application closely matches the functionality required in the new application, then, most extensions, be it additions or modifications, will be confined to the application classes. This has the direct benefit of reusing the effort originally required to understand the abstract classes' protocol[4] and specialization requirements.
- After reusing an application's application specific classes, it will be necessary to remove those classes which are not used.

Current programming environments a la Sniff [Bis92], display class hierarchy boundaries transparently, i.e. without a clear separation between application and underlying framework classes. In ET++, although we were tempted to assume that the application source code was within separate directories of the underlying framework source, we had no guarantee of this. The same is true independent of the

[4]Protocol: the set of messages that can be sent to class instances.

underlying framework being used and therefore one should avoid making such naive assumptions. We were looking for a safe, general way to identify *Draw*-specific classes.

Our first, simple unstudied approach, was to locate all abstract classes. To do so, we proceeded with the knowledge that an abstract class is a class with at least one operation left unimplemented. Because some operations are unimplemented, an abstract class has no instances and is used only as a superclass [McG92]. Our hypothesis being that abstract classes are members of the underlying framework's set of classes only, whereas concrete classes belong to the application's set of classes. We soon, with experience, obviously realized that we could not be guaranteed that an abstract class was not part of the application's set of classes.

Our second approached resulted in the general, clean solution we had initially set out to find. We assumed that by comparing the accompanying application's class hierarchies and determining the class intersection set, this set would include only those classes belonging to the underlying framework. There is however the possibility that different applications have classes with the same name at the same relative hierarchy position. We therefore had to ensure that the number of applications was sufficiently large, reducing the probability that all applications had a class with the exact same name and at the same position in their respective hierarchies.

More formally then, we define a class hierarchy, **H**, as a triple (N, D, S), where N is a set of class names,

$$\mathtt{D}:\mathtt{N} \rightarrow \textit{class-descriptions}$$

is the class description function specifying the description of each class, and

$$\mathtt{S}:\mathtt{N} \rightarrow \textit{seq}(\mathrm{N})$$

is the superclass function specifying the sequence of immediate superclasses of each class [OH92].

The underlying classes of a framework are obtained by the repeated application of a hierarchy combination operator, "$\bowtie$", to the set of classes in each application class hierarchy.

$n_1 \bowtie n_0$ requires $n_1 \cap n_0$ and that is exactly what we are looking for.

Ossher and Harrison [OH92] have proposed tools for the combination of class hierarchies. We claim that such tools could be extended to include underlying framework class extraction, as we have proposed here.

Without reusing the *Draw* application, we would have lacked a guided design for the required shape classes and would not have been able to implement the behavior, detailed in the previous section, with the ease in which we did. We agree with those who argue that understanding someone else's code can be a considerable intellectual challenge. However, if the code is well written and consistent, this effort will soon be absorbed, and reading the "not invented here code" might eventually require little more effort than reading one's own. ET++ (and its applications such as *Draw*) were well written with a consistent style throughout. We soon found ourselves comfortable in reading the underlying framework and applications' code.

Our experience agrees with that reported in the *Genesis* project [RGP88] in that it is generally a good idea to reuse an application, hence its code, even if time and changes are required in its understanding, reuse and/or maintenance. The potential offered through such controlled code reuse, i.e. through the use of a reliable application, is exciting, to say the least.

4.4 Reuse-in-the-medium - Micro-Architectures

We should now be in agreement that applications' frameworks reuse both design and code. Some aspects of a design, such as the kind of objects, are easily described by code. Other aspects, however, such as the interaction among groups of objects, are not expressed well as code. This makes frameworks harder to understand than for instance abstract classes.

We therefore need a level of abstraction between that of code and frameworks themselves. Hence we introduce micro-architectures, the lower level of reuse-in-the-medium, abstracting objects and their interactions. As frameworks codify design knowledge of a particular domain, objects and their interactions codify design knowledge in terms of the behavior of object collaborations. We refer to this type of reuse as "micro-architecture" reuse. A framework describes the architecture of a system instantiated from it, hence micro-architectures can be considered just that, i.e. micro-architectures of the larger framework architecture. Micro-architectures reuse both the

design and code describing the kinds of objects and the interactions and control flow among them.

Micro-architectures are a way to abstract and to reuse design experience. According to Coad [Coa92], there are object-oriented design structures that emerge repeatedly in the development of frameworks. These structures, micro-architectures, are of course known by the designer(s), but unfortunately by very few others.

The situation depicted in figure 4 is that of a framework user unaware of the collaborating objects and their responsibilities. We experienced this same void of the underlying micro-architectures in using ET++. We did eventually discover and understand some of them but only after a great deal (and I mean GREAT) of hard work and perseverance.

Providing designers and application developers with a set of micro-architectures would:

- provide a common vocabulary for design;
- reduce system complexity by naming and defining abstractions consequently reducing a framework's learning time;
- provide building blocks from which more complex designs can be built, for example, ET++ itself;
- provide a target for the refactoring of class hierarchies (cf. 4.2.1).

We shall introduce new techniques, design patterns [GHJV93a, Coa92] and contracts [HHG90, Hol92], for describing high-level design. Unlike frameworks expressed in a programming language, these techniques depend on a special purpose notation. We shall now describe these techniques and provide examples extracted from *Macrotec*.

4.4.1 Design Patterns

In football, a pattern is a series of sprints, turns, crossings and twists applied in an overall strategy for improving field position. In architecture, a pattern is an

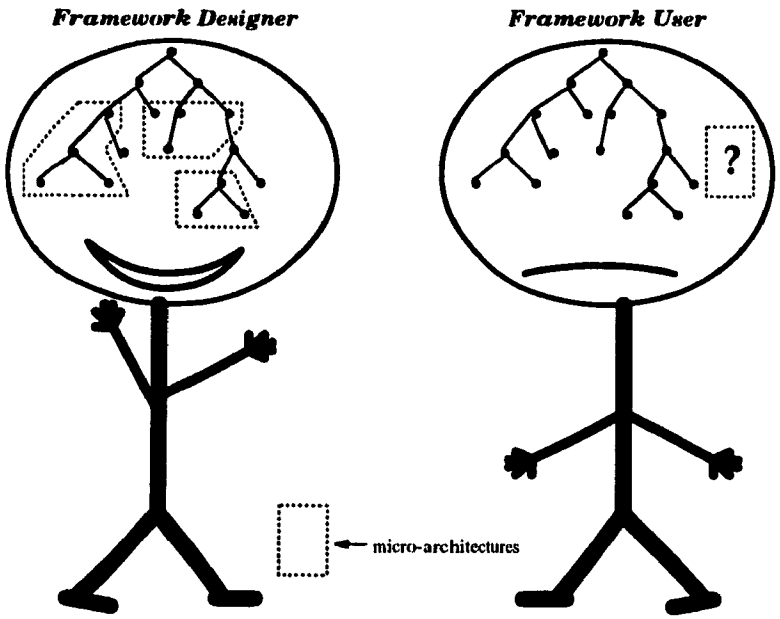


Figure 4: Framework Conceptual View - Designer versus User

architectural design or style. An architect, Christopher Alexander, was one of the early pioneers in design pattern theory [Ale79], "Indeed each building and each town is ultimately made out of patterns in the space, and out of nothing else; they [patterns in the space] are the atoms and molecules from which a building or town is made."

Similarly, each framework and its derived applications are ultimately implemented out of micro-architectures, they are the atoms and molecules from which a system is designed and built. Design patterns, as presented by Gamma in [Gam91], are a new way to identify and name object-oriented micro-architectures. They are a mechanism for expressing how components interrelate, a high-level representation technique for properly capturing and expressing design experience and intent to ultimately facilitate design reuse.

A design pattern consists of three parts:

1. An abstract description of a class or object collaboration and its structure. The description is abstract because it concerns abstract design, not a particular design;

2. The issue of system design addressed by the abstract structure. This determines whether the design pattern is applicable;

3. The consequences of applying the abstract structure to a system's architecture. These determine if the pattern should be applied with respect to other design constraints.

A designer familiar with a large set of design patterns (micro-architectures) can apply them immediately to design problems without rediscovering them. We instead, in developing Macrotec, reused many design patterns, without initially understanding the internal protocol and functionality. It was not until receiving a catalog of design patterns [GHJV93b] that we realized we were indeed dealing with pre-designed micro-architectures. This consequently helped us improve our design and, more importantly, lead to the understanding and hence resolution of some run time errors we had previously been unable to deal with. For example, we were unable to understand why, when the position of a graphical shape, e.g. an Action shape, was changed, its identifying label would update its position to that of the shape in question, namely to a default position at the upper left hand corner of the Action shape. Sure it was working the way we would have wanted to implement it ourselves, yet we felt uncomfortable not knowing the internals, i.e. the collaborating objects and underlying protocols. Almost unavoidably, an eventual change in requirements requested that we update the label's position such that it remained in the same relative position with respect to that of the graphical shape, i.e. no longer adjust the label to the default upper left corner. At first, we had no idea how to proceed, but then the *Observer* design pattern became available to us. The *Observer* pattern clarified how change propagation was designed in ET++ and thankfully helped us identify the necessary classes and methods for implementing the new requirement.

Other cataloged design patterns we found ourselves referring to during implementation included *Factory Method*, *Command*, and *Cookie*. We refer the reader to the catalog [GHJV93b] for a description of these patterns.

In developing Macrotec's graphical editor component, we had to design and implement the object collaborations necessary for graphical shape connections. We proceeded in an iterative way and decided to generalize our solution, describing it via a design pattern, the Connection pattern. The behavior described by the pattern is that which is required in connecting two graphical shapes (see figure 5). The pattern is meant to support maintainers of Macrotec as well as designers of other systems that have a graphical editor component. It is powerful enough to encompass editors with many different shape types that can be connected via various connection shapes.

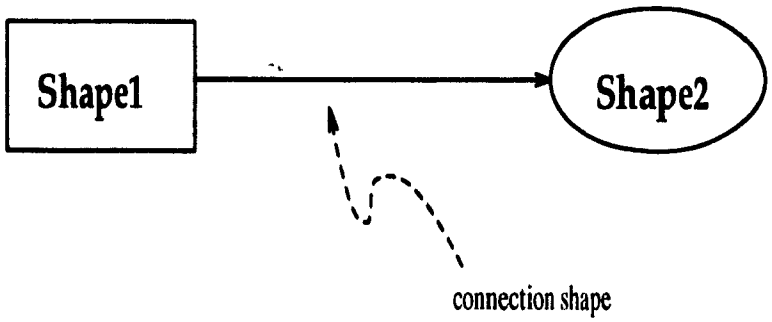


Figure 5: A connection between two graphical shapes

For describing the Connection pattern, we slightly modified the design pattern template suggested in [GHJV93b]. The modified template is depicted in figure 6.

DESIGN PATTERN NAME

The name of the design pattern is very important. It should clearly convey its intent. As this name will become part of the design vocabulary, it must be chosen carefully.

Rationale/Intent

What does the design pattern do? What is its rational and intent? What particular design issue or problems does it address?

Category

What is the classification of the pattern?

Motivating Example

A scenario in which the design is applicable, the particular design problem or issue the pattern addresses, and the class and object structures that address this issue. This example will help the reader understand the more abstract description of the pattern that follows.

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Description

Describe the objects participating in the design pattern, their resposibilities, and collaborations.

Diagram

The graphical notation based on the Object Modeling Technique (OMT) is used. The notation gives a compact and language independent view of a design pattern. Additions to this method include explicit object references and method pseudo-code.

Discussion

What are the tradeoffs and results of using the pattern? What does the design pattern objectify? What aspect of the system structure does it allow to be varied independently?

Implementation

What traps, pitfalls, hints, or techniques should one be aware of when implementing the pattern?

Contract Examples

This section lists example contracts from real systems.

See Also

What design patterns have closely related intent?
What motifs (cf. Chapter 6) describe issues involved in the pattern?

Figure 6: Design pattern template

In our description of the Connection pattern, we refer to the Command pattern which is itself described in greater detail in [GHJV93b]. In short, the Command pattern relates how commands decouple the creation of a request from the execution of the request. A command objectifies the request for a service.

Using the above template (based on [GHJV93b]), we came up with the following description of the Connection pattern.

CONNECTION Design Pattern

Intent This pattern allows lines, i.e. connections to be created between graphical shapes. It extends the *Command* design pattern. It lets graphical shapes treat a connection request in their own way decoupling the creator of the connection request from the executor.

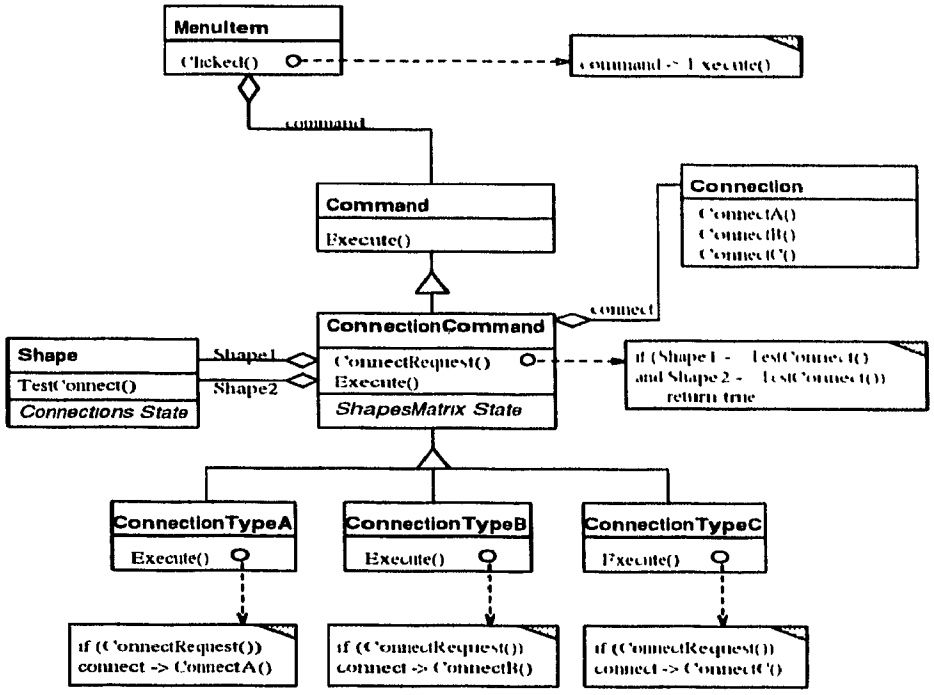
Motivation Used whenever connections are required between graphical shapes. A data structure is required to maintain connection rules which are used to determine if and what type of a connection between two shapes is legal (*ShapesMatrix State* in the diagram below). The key to the Command pattern is the abstract Command class. Subclasses of Command store the target of the request, i.e. **Connection**, and invoke one or more specific requests on the target.

Applicability See Command pattern. When independence on the particular connection request is required. When dealing with a graphical editor encompassing many different shape types that can be connected via various connection shapes.

Participants :

Command declares a generic request protocol for executing the command. Subclasses define and implement this protocol and maintain a reference to a command target, **Connection.** Subclasses request the specific service from the command target and add state they need to carry out the request.

ConnectionCommand abstracts the verification of whether or not a connection between the two shapes is legal. The state information, *ShapesMatrix*, will have to include specific information for each graphical shape pair which may be connected.



ConnectionTypeX responsible for instantiating the connection type and calling the proper method in the target connection.

Connection (target) the target is a connection shape and is likely a subclass of a graphical shape for drawing a line. A method for each different type of connection is recommended.

Shape Each shape involved in the connection is polled with relevant information regarding the proposed connection. Each shape will determine if the connection requested is permitted. The state should maintain information, *Connections State* relevant to each specific connection type the shape permits as well as information as to those connections it is already involved in.

Collaborations The subclasses of ConnectCommand will each represent a different type of connection i.e. arrow head, dashed, etc. The graphical shapes involved in the connection are each polled with the relevant connection information. Each shape will then determine if the requested connection is permitted.

Discussion The state information to be maintained by the ConnectionCommand, should contain data such as the number of inputs/outputs for connections, the

connection begin and end positions, etc. The state maintained by each graphical shape should be used by TestConnect() to determine whether connections are permitted. This information is specific to each graphical shape.

Implementation The target, connection, could be parameterized in order to avoid creating multiple Connection subclasses.

Contract Examples *Connection* contract from an ET++ application.

See Also

Design Pattern: Command (here we are simply extending the Command design pattern)

Motif:DRAW Connections between Graphical Shapes (cf. section 6.2)

Note that, to be included in their catalog, Gamma et al. require that a design pattern be representative of good object-oriented design and have a real practical application history in at least two different domains. We feel that the Connection pattern introduced here is of good object-oriented design and is useful for our own purposes. However, since it has not been reused yet in other domains, it does not qualify yet for inclusion in the catalog.

Our first experience with a design pattern was for the purpose of clarifying an implemented design in the *DRAW* application, specifically, that of *Factory Method* design pattern. In analyzing the code we realized that there were collaborating objects, however we did not fully understand the reason why such a design had been used, i.e. its intent. It was quite obvious that the design in question was non trivial and had to have been carefully thought out. We therefore suspected it to have a describing design pattern, yet which one? We had absolutely no idea which pattern specifically applied, if any at all for that matter. Upon studying the above mentioned catalog, we realized that the design under consideration matched with the factory method.

The above example illustrates two major prerequisites for the successful use of design patterns, namely, a framework and/or applications whose design is based on design patterns and a catalog describing those design patterns. Once these prerequisites are satisfied, we see three major ways to make use of design patterns:

Blueprint - as a guiding blueprint to introduce a new design;

Reference - as a reference to ensure that hierarchy restructurings and/or class redesigns do not affect behavioral collaborations;

Understanding - as a means to understand the underlying framework and/or application(s) developed from it.

Pattern identification remains as one of the fundamental problems in using design patterns for the purpose of **Understanding** and **Reference**. We therefore suggest that new design implementations include an identification of the guiding pattern in the component classes. This would allow one to easily identify the correct pattern. This is unfortunately not the end of the story, for after having identified the pattern, it is likely that one realize that there are more components involved in the collaborations than originally identified. This would therefore require performing the opposite identification assignment i.e. identify the rest of the implementation components involved in the behavioral composition. To this end, when introducing new designs guided by a pattern, **Blueprint** use, the application class and method names involved in the pattern should incorporate the class/method name as given in the design pattern. This has the advantage of making the design pattern and hence the involved classes and methods easily identifiable. Comming back to the Factory Method, we are convinced that, if this had been the case in the cited example, we would have much faster identified the design as an instance of the Factory Method pattern.

These simple suggestions would allow one to quickly reference the relevant pattern(s) and subsequently all involved components, resulting in a much better understanding of the design implementation as well as an overall reduction in the time required to do so.

We agree with Gamma that true design patterns are suppose to be generally applicable. However, we strongly suggest that design patterns be developed for framework-specific designs as well. Framework specific designs patterns will possibly support the design of numerous applications built on top of the particular framework and thus yield a big payoff.

We are currently elaborating ET++-specific behavioral compositions into such framework-specific design patterns. For instance, ET++ has its own unique start-up requirements for each application. The *main* program creates an *Application* object

and calls the *Run* method for that object in order to hand over control to ET++. This general start-up behavior is depicted in figure 7, which is a excerpt from our *ET++ start-up* design pattern. the figure also shows the internal behavior of the Application class, depicting the internal message passing path until the DoMakeDocuments message is eventually sent. This pattern has already proven useful. It helped two people who recently joined our group, with no prior knowledge of ET++, to quickly understand ET++'s start-up workings.

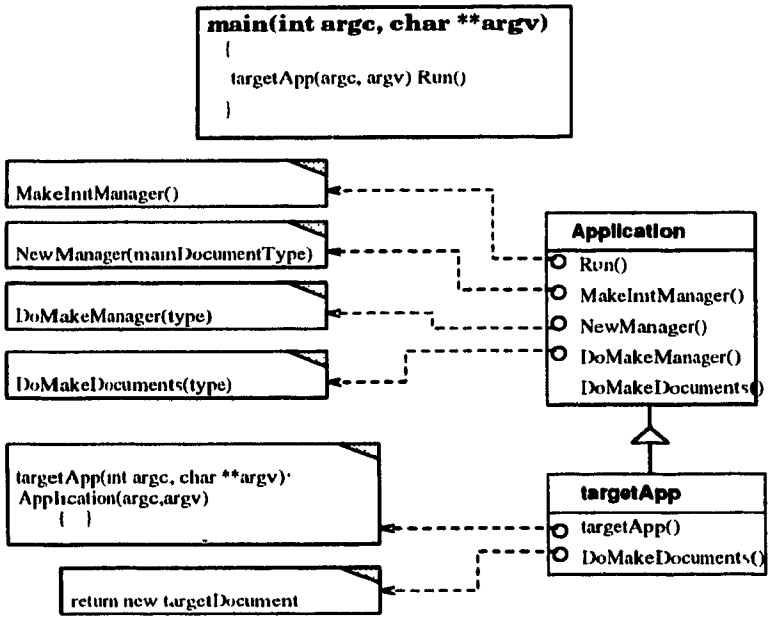


Figure 7: ET++ start-up behavior for an application, targetApp

Design patterns tend to be difficult to understand in isolation, mainly because of their high level of abstraction. This high-level of abstraction is however intentional and indeed required in order to ensure wide applicability. A more detailed form of expression would not be optimal for reuse, since it would be difficult to separate the individual design factors, i.e. the involved components: classes, methods, instance variables. Consequently, it would be difficult to understand and adapt a pattern to future design considerations. Abstract design patterns allow designers to edit, combine, remove, etc. the involved components and to come up with new design implementations.

Design patterns become much clearer, when they are accompanied by example implementation code. We believe that examples, as described by Contracts (see next

section), help designers understand some of the intended details behind the design. For this reason we have modified the design pattern template originally proposed by Gamma et al. to include the section **Contract Examples**. This section is intended to make reference to different Contracts developed from the design pattern in question. Below, we will introduce Contracts and describe a Contract developed from the *Connection* design pattern.

4.4.2 Contracts

Design Patterns describe framework design at a very high level. To ease the derivation of concrete design from them, an intermediate representation is required. Often these intermediate representations are called Contracts [HHG90, Hol92].

A Contract is simply a construct for explicitly specifying interactions among groups of objects. Recent literature recognizes the importance of object behavior collaborations [WBJ90] and responsibilities [WBWW90]; contracts formalize these collaborations and behavior relationships.

There is a definite lack of consensus amongst the involved researchers as to contract formalisms as well as to their level of abstraction. There have been many attempts to describe contracts. Attempts to use special-purpose programming languages have not been successful to date [Hol92]. Most other approaches, including our own, are based on informal notation, mostly some form of pseudo code. The Contract template presented in figure 8 illustrates our contract description technique. Each contract begins with cross-references to the pertinent design pattern and/or *motif(s)*. Then, a set of participants (classes) with each participant having its own set of contractual obligations is given. These obligations can be both type obligations, where the participant must support certain variables and external interface, and causal obligations, where the participant must perform an ordered sequence of actions. Also, contracts may define invariants that participants must maintain when cooperating. Finally, contracts may specify preconditions on participants to establish the contract and the methods required for the instantiation of the contract.

We have maintained the original theory behind contracts, however we have not strictly adhered to their (varying) formalisms. Ours is more of a pseudo-code style with adaptations for the support of our framework description techniques, namely

CONTRACT NAME

The name of the contract is very important. It should clearly convey its intent.
As this name will become part of the design vocabulary, it must be chosen carefully.

Pertinent Design Pattern and Motif(s)

Identify the design pattern abstracting the contract's design i.e. if of course one exists.
Identify the motif(s) within which this contract is involved i.e. if any at all.

Participants

A participant may either be Active or Non-Active.
Non-Active participants support type obligations only.
Active participants support both type and causal obligations thus maintaining
Non-Active participants.

Participant(s) Pseudo Code Details

Each participant details all type and/or causal obligations i.e. the class data members and methods (in pseudo code) involved in the contract.

Invariants

Definition of the invariants that participants cooperate to maintain.

Instantiation

Indentification of the preconditions necessary to establish the contract as well as the methods for its instantiation.

Figure 8: Contract template

those of design patterns and *motifs* (cf. chapter 6). Understanding how the participants interact via message passing (method calls) is the intent and importance of our contract descriptions. In reading a contract one should therefore not expend too much effort in understanding included implementation details. This being said, we are now ready to introduce a contract extracted from *Macrotec*. For the sake of brevity, we present the contract without its participant's method implementations. We refer the reader to appendix A for the complete representation.

CONNECTION Contract

Pertinent Design Pattern and Motif(s)

Design Pattern: Connection

Motif(s): none

Participants

```
Active:
 Command          : ConnectionCommandRequest;
 Shape            : Shapes;
 Shape            : ConnectionShape;
Non-Active:
 MatrixConnect in ConnectionCommandRequest
 Connection in MatrixConnect
 ShapeConnect in Shapes
```

Participants Pseudo Code Details

```
ConnectionCommandRequest supports [
    matrix : 2DArray(MatrixConnect)

    Legal(begin:Shapes,end:Shapes):MatrixConnect{}
    TrackMouseConnectRequest(begin:Shapes,end:Shapes,
                                        connectType:int):boolean {}
    BeginExecute(begin:Shapes,end:Shapes,connectType:integer){}
]
Shapes supports [
   value        : Value
   Coll         : Array(ShapeConnect)

   GetValue():Value { return value }
   TestConnect(connTypeInfo:Connection,other:Shapes):boolean {}
   Search(conntype : integer,ShapeValue : integer):ShapeConnect {}
   Create(conntype : integer, ShapeValue : integer): void {}
   GetConnPos(Origin:Point,Extent:Point,conntype:integer,
```

```
                                        ShapeValue:integer):Point {}
    UpdateIO(conntype : integer, ShapeValue : integer):void {}
]
ConnectionShape supports [
    NewConnection(start:Shapes,end:Shapes,matentry:MatrixConnect){}
    Draw(BeginPoint : Point, EndPoint : Point){}
]
MatrixConnect in ConnectionCommandRequest supports [
    connect      : Array(Connection)
    begin-origin : Point
    begin-extent : Point
    end-origin   : Point
    end-extent   : Point
]
Connection in MatrixConnect supports [
    conntype  : integer
    max-begin : integer
    max-end   : integer
    relShape  : GraphicalShapes
]
ShapeConnect in Shapes supports [
    ShapeVal : integer
    conntype : integer
    count    : integer
]
```

Instantiation

```
    ConnectionCommandRequest -> BeginExecute(begin:Shapes, end:Shapes,
                                             connectiontype : integer)
```

An Overview of the *Connection* contract

Notice how we identify the design pattern abstracting the contract's design, if one exists. There are three active participants in the contract: ConnectionCommandRequest, Shapes, and ConnectionShape. The three non-active participants MatrixConnect, Connection, ShapeConnect are data structures, i.e. participants with type

obligations only, maintained by other participants. We have introduced the idea of *Active* and *Non-Active* participants in order to allow Active participant's type obligations, i.e. Non-Active participants, to be expressed. We realize this adds implementation details however as long as such *state* descriptions are restricted to those identified in the design pattern, the understanding of both the design pattern and contract, and the relation between the two, should be improved.

ConnectionCommandRequest supports an important type obligation, *matrix*, a two-dimensional array structure of type *MatrixConnect*. The matrix is static and the information it maintains, MatrixConnect, is used throughout the contract, passed as arguments to other participants. The MatrixConnect structure is described at the end of the contract.

The contract is instantiated once the message *BeginExecute* is received by the ConnectionCommandRequest participant. This will lead to the sending of a message, *TestConnect*, to both graphical shapes involved in the connection. Depending on the result of the TestConnect call, *NewConnection*, could be sent to ConnectionShape for the purpose of creating the connection.

The Shapes participant, also supports an important type obligation, *Coll*, a single dimensional array structure of type *ShapeConnect*. The array is dynamic and the information it maintains, ShapeConnect, is used only within the Shapes participant. An array entry will be created for each different connection type and a counter for each will be maintained in order to keep track of the number of connections of each particular type. The ShapeConnect structure is described at the end of the contract. The method TestConnect is responsible for determining if the connection is possible at that time, for that particular "other" shape and connection type requested.

The NewConnection participant is fairly straight forward. It sends a message to each graphical shape, GetConnPos, requesting that they calculate the connection points for the drawing of the connection.

An important feature left to mention is that of contract *conformance*. Conformance declarations are specifications of how classes are eventually mapped to participants in a contract. Conformance declarations allow the typing and causal obligations of contracts, to be satisfied by the participants, to be distributed among the implementation of an abstract class and its subclasses. A conformance declaration must therefore declare explicitly which obligations are fulfilled by the abstract class and

which by the subclasses. This aspect of conformance declarations, understanding the implementation dependencies between abstract classes and their subclasses, is a feature that we believe could improve the understandability and hence use of abstract classes. The conformance declarations belonging to the *Connection* contract follow.

Conformance of *ConnectCommandRequest* participant

```
class ConnectCommand conforms to ConnectCommandRequest in Connection
        ConnectCommand supports
              Matrix of MatrixConnect
              TrackMouseConnectRequest(Shapes,Shapes)
        requires ALL subclasses to support
              BeginExecute(Shapes,Shapes)
end conformance
```

Conformance of *Shapes* participant

```
class Shape conforms to Shapes in Connection
        Shape supports
              Array of ShapeConnect
              GetValue()
              Search(conntype : integer,ShapeValue : integer)
              Create(conntype : integer, ShapeValue : integer)
              UpdateIO(conntype : integer, ShapeValue : integer)
        requires ALL subclasses to support
              TestConnect(connTypeInfo:Connect,integer:id,Shape:other)
              GetConnPos(Position : Point,conntype : integer,
                                        ShapeValue : integer)
end conformance
```

Conformance of *ConnectionShape* participant

```
class LineShape conforms to ConnectionShape in Connection
        LineShape supports
              Draw(Point,Point)
```

```
        requires subclass to support
              NewConnection()
end conformance

class Connection conforms to ConnectionShape in Connection
    inherits from LineShape;
        Connection supports
           NewConnection(Shape,Shape,MatrixConnect);
end conformance
```

Contracts are an intermediate representation between a micro-architecture and its corresponding design pattern. A contract can therefore, but not necessarily (refer to path **C** figure 9a)), be an intermediate development requirement when applying a design pattern to the creation of a new micro-architecture. It is, however, absolutely necessary to develop a contract as an intermediate stage in the development of a design pattern from a micro-architecture (path **C** of figure 9a) is unidirectional). The fact that we allow micro-architectures to be developed directly from a guiding design pattern, contradicts the usage proposed by Helm et al. where contracts are first-class objects in a new design paradigm, *interaction-oriented* design. Design then becomes a two-step process, where behavioral compositions are first defined via contracts which are in turn factored into class definitions and hierarchies via the contract's conformance declarations. Originally, as mentioned in the introduction of this section, a programming language was envisaged, for the definition and instantiation of contracts. Such a language would shift programming from the class level up to the interactions among the more abstract objects of contracts.

However, as Holland [Hol92] has noted, establishing a formal correctness criterion to verify conformance declarations at compile time is a difficult problem. Current support for the automatic generation of contract implementations may solve this verification problem, yet at the cost of shifting the burden of detailed control logic to the specification and hence to the designer instead of to the implementation.

Specifying a contract for the purpose of automatic implementation generation consequently requires that contracts be very detailed. In our opinion, these spec's would be too detailed and would resemble the actual code implementation in the framework. Thus the ultimate purpose of improving understandability, would be defeated. We seriously doubt that contracts will eventually be the basis of a programming language. We do however envisage contracts as a useful formalism to express high-level

specifications of object behavior, with emphasis on the minimum detail required to express objects' (participants') interactions.

Contracts used to describe frameworks in this way provide the application designer with:

- a vocabulary with which to describe the application;
- through conformance declarations, the identification of the application-specific classes, variables, methods, and hooks for customization, all necessary for identifying, maintaining and implementing a behavioral composition;
- knowledge of, and a better understanding of, individual micro-architectures present in the underlying framework, thus improving the understanding of the overall framework;
- guidance when refactorings affect participants in a contract, as for instance, during micro-architecture design iterations.

Contracts should not be taken as a means to understand the functionality of classes and methods. A class may participate in many contracts, its total functionality being separated into different contexts thus making it difficult to assimilate and understand. Also, methods described in contracts specify the minimum actions required, i.e. they may actually implement more than that described in a contract. Indeed, the same method may conform to more than one action body of a contract participant. This typically occurs when the class to which the method belongs to participates in more than one contract. If one wants to thoroughly understand a class and/or method, code inspection still remains the most precise and sure way.

4.4.3 "Design Patterns and Contracts in concert"

The previous sections described the techniques of design patterns and contracts, both used for the high-level description of behavioral compositions, micro-architectures. They are however different in terms of their level of abstraction in their descriptions. Design patterns are of course higher in abstraction. We do see both as describing interactions among groups of objects, however, a contract represents but one implementation/interpretation of a behavior composition. As depicted in figure 9 b), a

design pattern may be applied to many different contract/micro-architecture implementations, and we have a one to many (1:M) relationship. As an analogy, if I asked a group of people to write a code segment to reverse a single-linked list, the code segments would undoubtedly all be different yet their general structures would all be similar. Design patterns represent the experience and intent behind the design of a "broad" micro-architecture specifying only the major structures. This broad view may then be interpreted and refined to eventually form a design contract. A contract, on the other hand, describes one and only one micro-architecture, and conversely, a micro-architecture is the implementation of one and only one contract (1:1). It is therefore possible to have several contracts expressing the intended behavior of a single design pattern, thus underlining the expressive power of design patterns.

Figure 9 a) depicts possible development paths for contracts and design patterns.

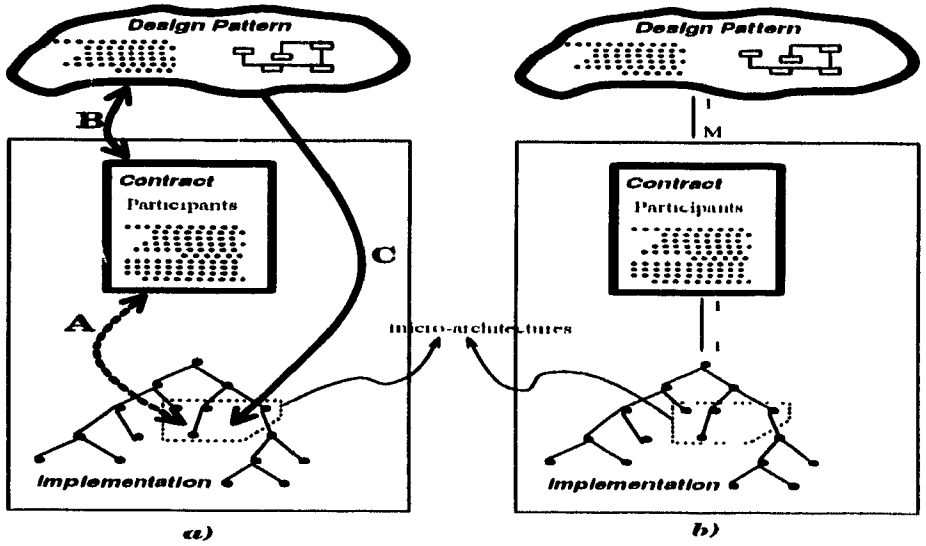


Figure 9: Description of micro-architectures: development paths **a)** and relationships **b)**

In developing a micro-architecture from a design pattern, there are two ways to proceed:

Directly, path C - To do so, fully understanding the intended behavior behind the pattern is essential. Therefore one will no doubt need to refer to previously described contract/micro-architecture pairs mentioned in the *Contract Examples* part of the design pattern template, refer to (figure 6);

Through an intermediate contract, path BA - The first step here is to try to understand the pattern with the help of previously described behavioral compositions i.e. other contract/micro-architecture pairs. The next step is to develop the pattern as a contract and factor it into the class hierarchy via conformance declarations. Then, as the implemented micro-architecture design iterates, the describing contract should be updated to reflect the design. This update during design iteration is depicted by the dashed arrow A in the diagram.

In developing a design pattern from a micro-architecture, there is only one way to proceed. There are however two distinct steps in the process. The first step is to describe the micro-architecture behavioral composition by a contract. In doing so, a designer may realize that the micro-architecture design could be improved. In fact, describing the contract will motivate the developer to go beyond concrete objects, i.e. it forces one to thoroughly understand object interactions and hence objectify concepts which are not immediately apparent as objects in the problem domain. Improving the micro-architecture design and subsequently updating the contract results in iteration as depicted by the dashed arrow **A** in the diagram. This iteration process should continue until a stable design is reached. If the designer then realizes that the behavioral composition in question has potential for broader applicability, the next step of describing a design pattern should be taken (arrow **B**). If not, however, there is absolutely nothing lost, i.e the process of describing the contract had obvious benefits. Also, a contract, in a contract/micro-architecture pair, with or without a corresponding design pattern, will have the same potential for use as that stated for design patterns, i.e. **Blueprint**, **Reference** and **Understanding** (see section 4.4.1).

It is important to note that not all contract/micro-architecture pairs will have a describing design pattern, in fact very few. A true design pattern will be non-trivial and will be applicable to several applications [GHJV93a]. This agrees with the claim made by Biggerstaff [BR87] that the broad structures (partial micro-architectures) are highly reusable and that the details typically are not. The broad structures must be, at a high-level, precisely described, while the details must be left incomplete and partially ambiguous. This is indeed supported by the one to many relationship between a pattern and the contract/micro-architecture pairs, indicating the high reusability of patterns.

We consider design patterns and contracts as valuable means to simplify and

guide existing object-oriented design methods [RBP+91] [WBWW90] [Boo91]. In fact, micro-architectures are fragments of a framework architecture and according to Waters [WT91], by reusing fragments of high-level designs, a software engineer can describe a program quickly and concisely. Therefore, it makes sense to describe and reuse design knowledge [ASP93] when,

- working on systems that are inherently difficult to understand such as very large systems and highly complicated applications, e.g. as those involving highly interactive graphical systems with intricate display requirements,
- working on systems with a high probability of design information being reused. These include the development of systems with a stable technology base as for instance object-oriented frameworks. All applications developed from an underlying framework have the potential of reusing much of the same design knowledge used in the development of other applications from the same underlying framework.
- working on systems with short product development life-cycles and/or those systems with long lives and/or those, known in advance, to require eventual future extensions i.e. evolving applications.

Lets conclude with a quote from Johnson, clearly supporting the design techniques mentioned throughout this section, "I don't believe drawing lots of pictures or diagrams is either design or analysis. I am perfectly happy understanding the design as abstract classes and interactions between objects. That's a model I can hold in my head, and when I look at code and interface files, I see that model realized. I don't really know how to draw pictures about it.", [WBVC+90].

Chapter 5

Reuse of "Black Box" Applications

Although reuse-in-the-large is not the main topic of this work, for completeness, we will briefly introduce some basic considerations and discuss our experiences as well as our experiences with this important level of reuse. We feel that the reuse of black box applications has considerably simplified the Macrotec development (cf. chapter 3).

5.1 Reuse-in-the-large

The highest level of reuse, reuse-in-the-large, is the reuse of objects which are themselves independent systems, systems which are reused as they are, without being modified or extended in any way. We call them "external systems". It is the system which reuses them, the "target system", which is responsible for adapting itself to the protocol requirements of the external system. The external systems may or may not have been developed with the same framework as the target system, they may even have been developed in a different programming language. The point to be stressed here is that the target system need not bother with the internal implementation and design details of the external systems since there is absolutely no intention of modifying or extending them.

Reuse-in-the-large involves problems typically encountered when integrating systems. To integrate an external system into a target system, first, interfacing requirements of the external system must be understood. In general transformation programs will then be required to transform information from the target system into the appropriate format of the external interface and, conversely, for extracting the needed information and transforming it back into the target system's format.

Reuse-in-the-large and system integration plays an important role in Macrotec. The approach we have taken is, according to Meyer's categorization [Mey91] of integration and extensibility, a blend of the simple database and, to a limited degree, the canonical representation approaches.

In *Macrotec*, we are using *Gemstone*[1], an object-oriented database management system allowing for the storage and retrieval of the core representation, see figure 1. The *core representation* is the heart of *Macrotec* with all information to and from the various tools, both internal and external, managed within. A weakness often cited with the database integration technique [Mey91] has been that the data structures supported by the database are in general not sophisticated enough to be used directly by the tools. Consequently, tools often retrieve the needed information from the database and build their own internal representation. At later points, they write out all the modified data all at once back into the database. This is, however, not the case in *Macrotec*. All Macrotec tools were written using the object-oriented paradigm, the C++ programming language, and may therefore directly use the class hierarchy, core representation, as defined by the Gemstone C++ interface, just as they would any other object-oriented class hierarchy.

According to Meyers, a canonical form is a single data representation, shared by all tools in an environment. Although this is not the case in *Macrotec*, we do have a single data representation for external tools which manipulate graphs. And, although not fully shared, in the sense of being directly supported by all the tools, our canonical form has indeed facilitated the integration of external tools. Typical canonical representations have some fundamental data structure for the core data and then let tool-specific data be added. Our canonical form, the *GXF+ representation*, is no exception. GXF+ is based on GXF [MEN92]. GXF allows graph manipulation programs to operate on data encoded in a common format which includes certain

[1]Gemstone is a registered trademark of Servio Corporation

essential features of graphs. In addition, GXF allows programs to add any arbitrary information to a GXF representation in a consistent, portable manner that is transparent to other programs which do not expect the extra information.

5.2 Application reuse in Macrotec

Figure 1 depicts the different tools involved in the *Macrotec* toolset. As "black-box" applications, we have integrated the *SPNP* performance analysis tool [TMWH92], an automatic graphic layout package being developed at the University of Toronto [MEN92], and a substitution tool developed at Laval University [JBB+92].

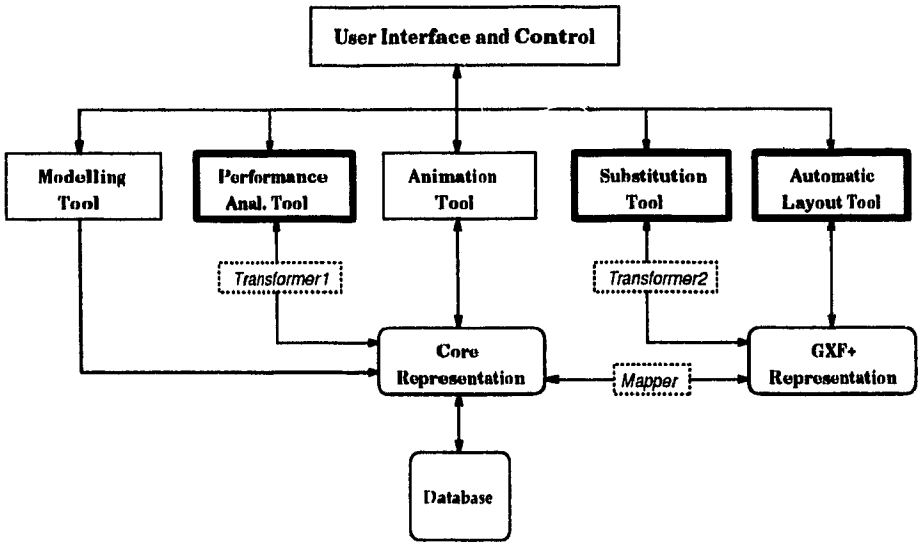


Figure 1: Macrotec Architecture Overview

The performance analysis tool manipulates non-graphical data and therefore does not deal with the GXF+ representation, but it requires a data transfer program to and from the core representation. We first had to study the *SPNP* system in order to determine its input file (*perfres.c*) and output file (*perfres.out*) formats. We then wrote a data transformation program, *Transformer1*, creating an input file to *SPNP* with the following three most relevant sections: *parameters* section storing the performance analysis parameters, *net* section describing the modelled network, and an *ac_final* section storing the results to be computed during analysis. The same

transformation program is responsible for extracting the relevant information from the output file (*perfres.out*) and transforming it back into the core representation.

The *Substitution* and *Automatic Layout* tools, manipulate graphical data. Such tools store their data in the *GXF+ representation*. As mentioned in chapter 3, supporting GXF+/GXF allows us to easily exchange data with other, special-purpose, GXF-based systems such as the automatic layout tools being developed at the University of Toronto. Non-GXF+-based systems require data transformation programs. For instance, integrating our substitution tool (implemented before adopting the GXF+ standard) required the development of the *Transformer2* program. Without the intermediate representation of GXF+, each graph manipulating external tool would require a transformation program between itself and the core representation. Consequently, a change in the database core representation may require a change to each of these transformation programs, obviously not the best of situations. Adopting the *GXF+ representation* results in *Mapper* being the only program requiring maintenance due to database core representation changes (significantly, the GXF+ representation is very stable as compared to the database representation). We hope that future graph manipulation tools will adopt the GXF representation allowing our system to be easily extended if ever the requirement arose.

There is not yet a consensus about the meaning of integrated system. We claim that ours is integrated, both externally, and at the user interface level, and internally. Although, as we have just described, transformations of all types are performed, at the user level, the *Macrotec* tools are seamlessly integrated i.e. the user interacts with the system through one single base window allowing for easy switching between the different tools. Internally *Macrotec's* loosely coupled architecture facilitates the addition of new tools. Adding new tools will not require that existing tools change their functionality or even be made aware of the addition.

Chapter 6

Prerequisites for Successful Reuse

Object-Oriented frameworks tend to be large in terms of the total number of classes. For example, ET++ contains more than 230 classes, the Interviews framework [LCV87], more than 150 classes. In addition to frameworks being huge monsters of classes, methods, and code, they typically come with little or no documentation. ET++ comes with virtually no documentation, whereas Interviews has comprehensive documentation about each class in the library but provides little information on how to use them. To fully exploit these vast amount of existing code for reuse, one must understand the underlying framework classes. Consequently, developing systems using object-oriented frameworks requires developers to spend much more time reading existing code (mostly other developer's code) rather than writing, often reinventing, new code.

We will now briefly discuss programming, design, and documentation issues we have found to directly impact the learnability, and hence usability of frameworks.

6.1 Learnability

Most developers would agree that reading another programmers code is a major intellectual challenge and one we practically all try to desperately avoid. Modern programming environments successfully support this process by providing facilities for browsing, cross-referencing, design visualization, editing, etc. However, they do

not, and will probably never, eliminate the need for code reading. Nevertheless, there are design and programming techniques that can considerably facilitate this underestimated task of understanding "not invented here" code.

Prior to *understanding*, we must *find* the desired reusable components. Many factors affect the ease with which developers can *find*, and subsequently *understand* components in a large framework. We believe that the most important ones are common vocabulary, number of levels in a class hierarchy, and component size (classes and methods). Each will be discussed in the following sections.

6.1.1 Common Vocabulary

An unfortunately common object-oriented programming practice is to use the same name for two unrelated purposes [ROL90]. This naturally leads to confusion when trying to understand as well as locate a component. The use of different names for similar objects, has resulted primarily from requirements of procedural programming languages that subprogram names be unique within a given program. Polymorphism provided by object-oriented programming languages eliminates the need to use different names for similar operations.What we want is a common vocabulary for similar entities, and different names for different entities.

The best known successes with component reuse have been in applications using standard data structures. Data structures have had certain names occur in implementations over and over again, for example, a first-in, first-out queue has two operations, usually called *enqueue* and *dequeue*, to add elements to and remove elements from the queue, respectively.

Abstract classes, especially the topmost levels, establish, by the very nature of inheritance and class hierarchies, a standard interface. The specification of standard methods, with standard method names, ensures that all subclasses will conform. The goal being to minimize the number of different names and maximize the number of names shared by a set of classes, i.e. standard protocols. For instance, the Collection class of ET++ provides a standard interface which includes operations Add, Remove, Find and many others. All subclasses of Collection (SeqCollection, OrdCollection, Set, etc.) use these names standardizing the entire range of collection classes.

The purpose of a method is to operate on an instance of a class or to provide information regarding the state of an instance. Methods constitute the interface of the class on which they are defined and should therefore reflect this orientation. For example, Point, is a typical class in many frameworks and provides methods to access the x and y coordinates. The name readX could be used, however, it would imply reading from a device. Naming the method getX would be much more descriptive. The suggested meaning of getX is more descriptive in itself but even more so due to its standard use and hence familiarization.

In naming components, one should keep in mind that names should give a reasonable indication of the point of view of the underlying design. Also, they should conform to the standard language of the domain. For example, getA instead of getX, is definitely not as clear due to "x" and "y" being so common in geometry and computer graphics domains.

Ultimately, a common vocabulary would be broadly applicable to multiple frameworks in multiple domains. Those components common to several different frameworks in multiple domains should share common names. The abstract class, Collection, should be named the same in all frameworks (ET++, Interviews, Smalltalk-80, etc.) that implement a set of container classes. A first step towards this ideal is the reuse of micro-architectures as described by design patterns (cf. section 4.4). Although the names that appear in a design pattern are typically too abstract to appear directly in an application, we suggest they be incorporated in the implementation names. For example, in using the *Observer* pattern for the design of change propagation, we would name methods DoObserve, RemoveObserve, etc.

The present lack of a common vocabulary has caused serious communication problems and has hampered the identification of components and thus the reusability of them.

A component which cannot be found, cannot be reused!

6.1.2 Number of Levels in the Class Hierarchy

A class description in an inheritance hierarchy, together with the class descriptions of its superclasses and their ancestors, define a class [OH92]. Defining a class hierarchy (refer to section 4.3 for details), **H**, as a triple (N, D, S), we have a formal definition

of a class. The repeated application of a class combination operator, "$\oplus$", to the class descriptions in an ancestor sequence, defines a class,

$$d_1 \oplus d_0 = d_1 \cup \{(s \mapsto m) \in d_0 \mid s \notin domain(d_1)\}$$

The definition of the class combination operator, "$\oplus$", is such that methods supplied by the left operand override identically named methods supplied by the right operand. Accordingly, the greater the number of levels in the hierarchy, the harder it is to determine the complete class definition. Shallow hierarchies make it easier to see which class implements what member function.

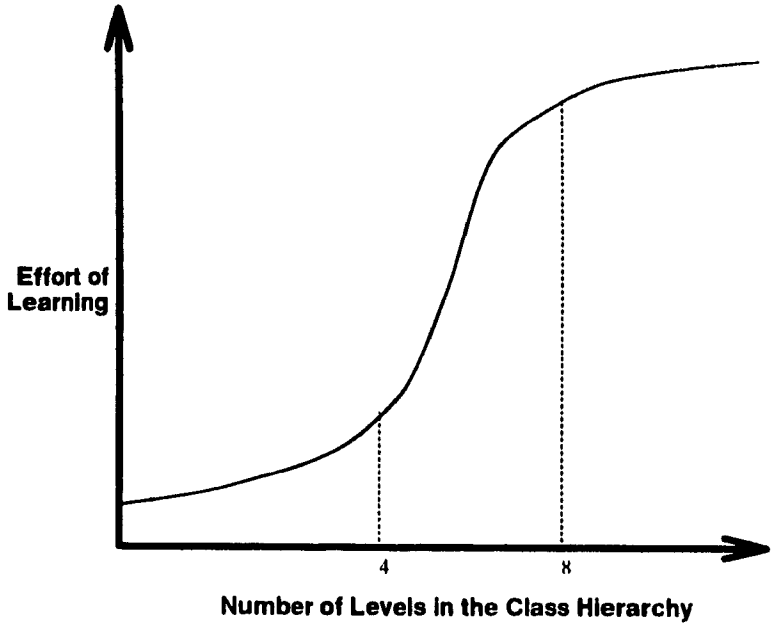


Figure 1: Number of hierarchy levels effect on Ease of learning

Figure 1 is an informal depiction of the effect of the number of hierarchy levels on the effort to learn the class hierarchy. The diagram reflects our experiences and contradicts *Rule 5* in *Rules for Finding Abstract Classes* proposed by Johnson and Foote [JF88] which states that class hierarchies should be deep and narrow. It does, however, agree with the developers of the Interviews library. The class library of Interviews was kept intentionally shallow (most classes are at level 2 or 3) based on their experience that many levels overwhelm programmers.

6.1.3 Component Size

As the number of classes in a framework increases, a greater investment is required in the effort to properly identify components. The identification of components in a framework has however been notably facilitated by the use of powerful object-oriented programming tools [Bis92] and other highly specialized techniques [PD91a]. Properly identifying a component does not necessarily require an exact match. It may simply require the location of similar components because for reuse, often only parts of an existing component are needed. This aspect of partial reuse reduces effort and improves reliability, as described in section 4.2 with the reuse of class BoxShape. The "close" matching of components is thus more directly affected by component size, and is a characteristic lacking satisfactory support in existing programming tools.

Experiments have shown that size is a significant factor affecting successful reuse [WES87]. As a class and/or method grows, it becomes increasingly difficult to reuse. The growth of a component causes it to become more and more specific and consequently narrows its applicability. A class with small methods is easier to subclass, since the behavior can be changed by modifying a few smaller (and hence easier to understand) methods instead of larger (and hence more difficult to understand) methods. We have found that a class with greater than 40 methods is in most cases too large and hence too difficult to reuse because of its specificity. Such classes should be redesigned and refactored into the class hierarchy.

As Biggerstaff has pointed out [FBPD+91], the importance of finding a component is closely related to the tension between size and reuse potential. Size is a metric that closely correlates with specificity, and specificity is the factor that veritably affects reusability.

During our early stages of development we found ourselves creating highly specialized classes therefore requiring that we have many of them, (all of them being very similar). Such a circumstance places even greater importance and difficulty in the finding (matching) process [WMH93]. An application will typically go through several maturity stages as depicted in figure 2. As design iterates, the class hierarchy is restructured. The restructuring process results in the maturing of a framework. As the classes and methods become fewer and smaller in size, it becomes relatively easy to understand each small components code, easier to find an appropriate match and consequently facilitates reuse. This therefore suggests that to practice software reuse, it is important to understand the stage of an applications maturity.

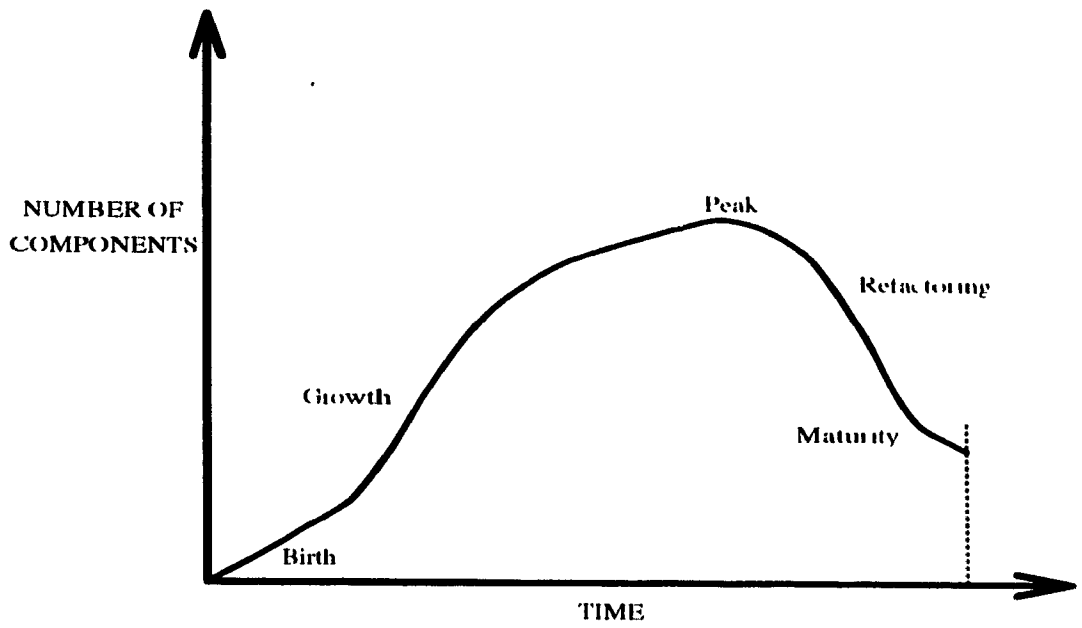


Figure 2: Application Life Cycle

6.2 Describing Frameworks

Documenting frameworks for reuse requires their description at different levels of abstraction, most importantly, at the design level. Furthermore, such documentation should address the needs of developers with varying levels of experience with the framework.

Novice users are interested in general, high-level aspects of the framework before dealing with the design details of micro-architectures. In other words, they require to know how to use the framework before knowing how it specifically works. We propose the use of a set of *Motifs* to show how to use a framework. The concept of motif was introduced by Johnson [Joh92] under the term pattern. We have slightly modified his ideas and adapted them to the abstraction levels of framework design descriptions proposed in section 4.4 (cf. Figure 9 a)). We have introduced the term **Motif**[1] in order to avoid confusion with *Design Patterns*.

The documentation of a framework should encompass:

- the purpose of the underlying framework;

[1] Motif (Merriam-Webster): 1. a dominant idea or central theme 2. a single or repeated design

- how to use the underlying framework;
- the purpose of application examples;
- how to reuse application examples;
- the design of the framework;

Motifs are responsible for describing the first four of the above list items whereas *Design Patterns* and *Contracts* address the last item, the design of the framework. *Design Patterns* deal with abstract designs, and *Contracts* cope with the detailed design of concrete examples.

Each *Motif* describes a situation which must be replicated in order to use the framework, whether it be the underlying framework or the framework of an application. All motifs have the same format (see figure 3). They begin with a title of the format, **Motif:***application name*. The *application* is the framework in question, i.e. the underlying framework such as ET++ or the framework of a developed application, e.g. *DRAW*. The *name* is simply the name most relevant to the situation described. Then, they briefly describe the situation (Situation), followed by a detailed discussion of how the situation may be adapted and/or utilized (Situation Discussion). A motif ends with references to other related motifs as well as to relevant design patterns and/or contracts.

A set of motifs could describe,

- the underlying framework, e.g. ET++. A first motif would describe the application domain of ET++ and list example applications developed from it. It would also list all other motifs which describe ET++ as well as a suggested order of reading.
- an example application, e.g. DRAW. A first motif would describe the application and its general functionality. It would also list all other motifs which describe the application as well as a suggested order of reading. This set of motifs should be written with a reuse theme. The motifs should focus on describing how the application in question can and should be reused. For example, the motif below describes how to introduce new graphical shapes into the *DRAW* application.

Motif:Application Identifier

The name of the motif is very important. It should clearly convey its usage.
As this name will become part of the design vocabulary, it must be chosen carefully.
Application - is the name of the framework in question.
Identifier - the name most relevant to the situation described.

Situation
A brief description of the situation is given.
Situation Discussion
A detailed discussion of the situation, clearly highlighting how developers should proceed in order to benefit from its understanding and reuse.
References
A list of related motifs as well as involved design patterns and/or contracts.

Figure 3: Motif template

Motif:DRAW New Graphical Shape

Situation

There are a variety of graphical shapes that can be incorporated in a graphic editor. Here we describe those shapes and how one goes about integrating them in an application.

Situation Discussion

Each graphical shape is a subclass of Shape. There are already subclasses of shape for the simple objects (LineShape, BoxShape, OvalShape, PolyShape, ImageShape, DynShape, PictureShape, TextShape), and these may in turn be subclassed to create more complicated shapes (RegionShape, BezierShape, ArcShape, RcBoxShape, Connection, DynShape2). The minimum required to define a subclass of Shape includes the definition of the methods Draw, Outline and GetImage.

Each application developed from DRAW will have a class *draw* whose constructor is responsible for setting up a palette of shapes. When adding a new shape, you are required to update the palette correspondingly. To do so, an image item (bitmap), to be displayed as the selection button, must be created and included in the new class's implementation, for example,

```
static short BoxImage{}={
#    include ''images/BoxShape.im''
};
```

The new shape class must then be added to the list of shapes in the palette, i.e. added to a collection called, *prototypes*. This is performed in the constructor of the class *draw*,

```
prototypes->Add(new BoxShape);
```

Most shapes will additionally define stretching functionality, input/output capabilities, selection handles, etc.

References

- Graphical shapes may depend on each other - see **design pattern**: *Observer*.
- Stretching capabilities - see **motif**: *Graphical Shapes Stretching Capabilities*.
- Graphical shapes may be connected - see **design pattern**: *Connection* and/or **Contract**: *Connection*.
- Complicated graphical shapes - see **motif**: *Subclassing Simple Shapes*

END — Motif:DRAW Graphical Shape

Unlike Johnson, we suggest that developers be required to know how a framework works. A developer aware of the underlying design, e.g. the micro-architectures as described by contracts and/or design patterns, will try to maintain the design intent during modifications ensuring that behavioral compositions are properly maintained. Motifs compliment design documentation. They do so informally without detailing algorithms or object collaborations. Instead, they often refer to design patterns and/or contracts, familiarizing the reader with important design details they should ultimately be aware of in order to preserve the design of the application and/or underlying framework. In general, a motif should be developed for any and all aspects of a framework, that potentially require adaptation or need clarification.

It appears that we have now covered all aspects in terms of framework documentation. Well, not quite. We need documentation support from the lower levels of a framework, the framework classes as well. Each class should have references to *motifs*, *design patterns*, and/or *contracts* it is involved in. In addition, whenever possible, the individual methods of a class should each identify the pattern/contract/motif within which they participate (recall that a class and/or method may participate in more than one micro-architecture). The class format should be similar to that given for the *BoxShape* class below.

```
#ifndef BoxShape_First
#define BoxShape_First
#include "Shape.h"
//---- Box Shape Participation ---------------------------------
// motifs ==> 1) motif:DRAW Graphical Shape
//            2) motif:DRAW Database Save/Load
// design patterns ==> 1) Connection
// contracts ==> 1) Connection
//-------------------------------------------------------------
class BoxShape : public Shape {
  SeqCollection *ExistingConnections;  // Contract Connection
public:
  MetaDef(BoxShape);

  BoxShape();
  short *GetImage();
  void Draw(Rectangle);
  void Outline(Point p1, Point p2);
  bool ContainsPoint(Point p);
  boolean TestConnect(Connection*, Shape*) // Contract Connection
//-------------------------------------------------------------
  virtual void DBLOADConnect (GPTR); // Motif:DRAW Database Save/Load
  virtual void DBSAVEConnect (GPTR); // Motif:DRAW Database Save/Load
};
#endif
```

Having classes reference their corresponding documents prevents implementors from inadvertently changing the responsibilities of a class that others may be gravely depending on. For instance, deleting a method thought to be no longer needed can be disastrous, e.g. an implementor would certainly be wise to look-up the Connection contract before deleting the TestConnect method (we now all know the important role it plays in the object collaborations for creating connections (cf section 1.1))

Although *motifs* are primarily aimed at describing how one should proceed in reusing a framework, their presence greatly adds to the descriptive power of design patterns and contracts. They clarify many details as to how the design is applied to an application as a whole

Chapter 7

Summary

Our experience with ET++ has confirmed that developing an application based on an object-oriented framework requires a substantial initial investment in learning. We required a full two and a half months of intense code inspection and hacking in order to feel comfortable enough with ET++ to begin the *Macrotec* development. The predominant reason for such steep learning curves is because design information is being lost and buried in the implementation code.

As Horowitz pointed out [HM84], one of the main inhibiting factors for the reuse of design is the lack of design representations that promote reuse. We have proposed *Design Patterns* and *Contracts* as design representations of micro-architectures. We have concentrated on micro-architectures (behavioral compositions, collaborating classes) and their underlying design rather than on classes. Although classes have been proposed as units of code reuse, a class often depends on others hence it is not single classes but groups of classes which are reused. In addition, we suggest the underlying framework as well as each example application derived from it, have their own set of *motifs*. *Motifs*, besides describing how one uses a system, may include references to all design-level descriptions involved, i.e. to all design patterns and contracts.

To successful reduce the learning curve, the framework description techniques proposed, design patterns, contracts, and motifs, along with the corresponding code components, classes and methods, must cross reference each other whenever appropriate. There are many advantages to describing and organizing frameworks this way, including:

- the retrieval of reusable components (micro-architectures and/or classes and methods), is facilitated
- the effort required to understand is reduced. For example, when trying to understand an implemented behavioral collaboration, having references to abstract design descriptions such as contracts and/or design patterns is a definite blessing.

However, in order to avoid creating an overwhelming web of cross-references, the developers of frameworks and applications have to follow two basic guidelines (which express nothing more than good object-oriented design) The first is to minimize the number of collaborations a class has with other classes, and the second is to minimize the number of different *contracts* supported by a class

In short, our description techniques decrease a framework's learning curve and consequently increase a frameworks reusability. They will however require that the design be oriented towards intermediate abstraction levels and that this kind of design become an integral part of the software process. The initial, additional cost imposed by this approach can be amortized over the products developed from the described reusable design. This added investment during the software process, is a worthy one indeed.

7.1 Advantages of using Frameworks

During the development of *Macrotec* we have experienced several advantages of using frameworks.

- Through polymorphism and inheritance (with frameworks, inheritance is an important feature for code reuse), we experienced significant *code reuse*. For example, the use of virtual functions in C++ allows many methods to be reused.
- The reuse of design patterns permits *design reuse* not only within a framework but among different frameworks as well. For example, the Observer design pattern, used in ET++ for the implementation of change propagation, is also used in Interview's *Unidraw* [VL89].

- Frameworks further system and machine portability by hiding dependencies within classes. For instance, the two abstract classes *System* and *Window System* are responsible for instantiating new objects representing window system resources. The system-dependent parts are specified by pure virtual functions. Presently, ET++ runs under SunWindow, NeWS or the X11 window system.

- Code and design reuse reduce coding and design time, respectively. This encourages developers to try out new ideas. In other words, frameworks encourage *rapid prototyping*.

- Frameworks facilitate the coordination of people working on the same project. Work can be divided into different parts of the framework hierarchy where changes to classes are independent.

7.2 Future Research

Ideally, the above mentioned cross references would be supported by a hypertext system providing a mental model of the links between components. Hypertext systems such as *PlaneText* [Gea86] allow for webs of such cross reference information that smoothly integrate text, graphics diagrams, and code. Thankfully, the PlaneText system allows existing code files to be annotated without being altered.

Figure 1 depicts a typical web of cross references between code, design descriptions and motifs. A framework complemented by such a system would improve retrieval and understandability by providing instant access to supporting information.

As mentioned, contracts help in the understanding of micro-architectures. To understand how micro-architectures themselves interact, we have motifs, providing a textual explanation. Textual explanations are fine but often too verbose and ambiguous. We would need to supplement motifs pictorially. Buhr and Casselman [BC92] have proposed the use of timethreads as a visual notation to communicate the interaction among contracts. Pictures, as shown in figure 2, would no doubt be useful to anchor the concepts presented in motifs.

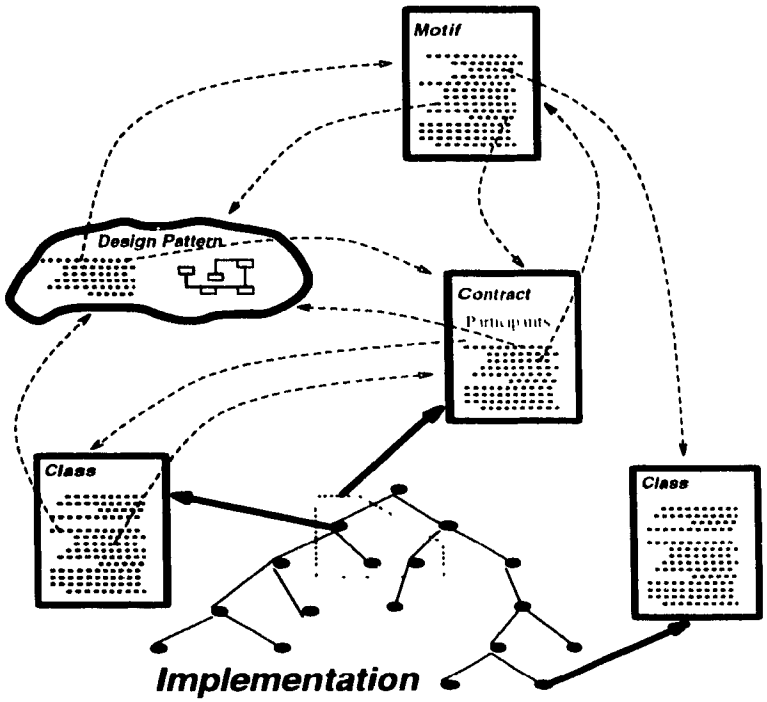


Figure 1: Example cross reference links between components

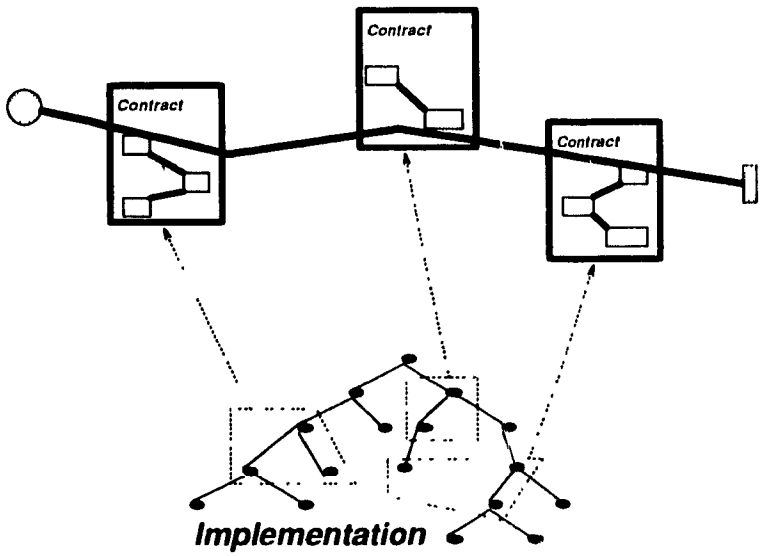


Figure 2: Timethread that spans the contracts of a framework

7.3 Concluding Remarks

We are not claiming, with the contents of this thesis, to have created a panacea. It is clear that considerable work remains to be done in order for eventual catalogued specific frameworks with wide spread practical applicability. We have however provided a first step, a step we believe to be in the right direction. To successfully use and reuse frameworks, it is now up to us to become a new breed of developers, developers with a new attitude towards using "not invented by me code", developers, to quote from Beck [OBHS86], "with an obsession for simplicity, who are willing to rewrite code several times to produce easy-to-understand and easy-to-subclass classes".

Just as the Eskimo has many different words for snow, we have many words for reusability. A plausible conclusion is that reusability of framework components is (or should be) as important in our lives as snow is in the life of the Eskimo.

Appendix A

CONTRACT - Macrotec Constraints

The *Connection* Contract

```
contract Connection
  DESIGN PATTERN : Connection

  participants
    Active:
     Command          : ConnectionCommandRequest;
     Shape            : Shapes;
     Shape            : ConnectionShape;
    Non-Active:
     MatrixConnect in ConnectionCommandRequest
     Connection in MatrixConnect
     ShapeConnect in Shapes
  ConnectionCommandRequest supports [
      matrix : 2DArray(MatrixConnect)

      Legal(begin:Shapes,end:Shapes):MatrixConnect{
        return m at location (begin->GetValue(),end->GetValue()) in matrix
       }

      TrackMouseConnectRequest(begin:Shapes,end:Shapes,
                                        connectType:int):boolean {
        MatrixConnect : entry
        entry = Legal(begin,end)
        if (entry)
```

```
            if ((begin->TestConnect(entry.connect[connectType],end)) and
                    (end->TestConnect(entry.connect[connectType],begin)))
               return true i.e  connection permitted
            else
               return false i.e connection NOT permitted
         else
           return false
      }
      BeginExecute(begin:Shapes,end:Shapes,connectType:integer){
          if(TrackMouseConnectRequest(begin:Shapes,end:Shapes,
                                                  connectType:int))
            return ConnectionShape.NewConnection(begin,end,entry)
          else
            Refuse the connection!
      }
]

Shapes supports [
   value         : Value
   Coll          : Array(ShapeConnect)

   GetValue():Value { return value }

   TestConnect(connTypeInfo:Connection,other:Shapes):boolean {
     ShapeConnect : ShapeConnInfo;
     if ((ShapeConnInfo = Search(connTypeInfo.conntype,other->GetValue())))
       if (other is END shape)
         if(connTypeInfo.max-begin == (ShapeConnInfo.count)
            return false
         else
            UpdateIO(connTypeInfo.conntype,other->GetValue())
            return true
       else
         if(connTypeInfo.max-end == (ShapeConnInfo.count)
            return false
         else
            UpdateIO(connTypeInfo.conntype,other->GetValue())
            return true
     else
       Create(connTypeInfo.conntype,other->GetValue)
       return true
     endif
   }
```

```
   Search(conntype : integer,ShapeValue : integer):ShapeConnect {
     forall c in Coll{
            if (c.conntype == conntype AND c.ShapeVal == ShapeValue)
               return c
           }
           return 0
   }

   Create(conntype : integer, ShapeValue . integer)· void {
        Coll->Add(conntype,ShapeValue,0)
   }

   GetConnPos(Origin:Point,Extent:Point,conntype:integer,
                                   ShapeValue:integer):Point
   {
      return position according to Coll.conntype,Coll.ShapeVal
   }

   UpdateIO(conntype : integer, ShapeValue : integer) void
   {
      At entry in Coll with same conntype,Shapevalue --
                    Coll.count = Coll.count + 1;
   }
]

ConnectionShape supports [

   NewConnection(start:Shapes,end:Shapes,matentry:MatrixConnect)
     {
         Draw(start->GetConnPos(matentry.begin-origin,
                                matentry.begin-entent,
                                matentry.conntype,end->GetValue()),
              end->GetConnPos(matentry.end-origin,
                              matentry.end-extent,
                              matentry.conntype,start->GetValue()))

     }

   Draw(BeginPoint : Point, EndPoint : Point)
     { draw line shape }

]

MatrixConnect in ConnectionCommandRequest supports [
```

```
   connect       · Array(Connection)
   begin-origin  · Point
   begin-extent  : Point
   end-origin    : Point
   end-extent    : Point
]

Connection in MatrixConnect supports [

   conntype   : integer
   max-begin  : integer
   max-end    : integer
   relShape   : GraphicalShapes
]

ShapeConnect in Shapes supports [

   ShapeVal  : integer
   conntype  : integer
   count     : integer
]

instantiation
   ConnectionCommandRequest -> BeginExecute(begin:Shapes, end:Shapes,
                                            connectiontype : integer)
```

Bibliography

[Ale79] Christopher Alexander. *The timeless way of building*. Oxford University Press, New York, 1979.

[ASP93] Guillermo Arango, Eric Schoen, and Robert Pettengill. A process for consolidating and reusing design knowledge. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 231–242, Baltimore, Maryland, May 1993.

[Bar84] J.G.P. Barnes. *Programming in Ada*. International Computer Science Series. Addison-Wesley, Menlo Park, second edition, 1984.

[Bas90] Victor R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25, January 1990.

[BC92] Raymond J.A. Buhr and Ronald S. Casselman. Architectures with pictures. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 466–483, Vancouver, B.C., October 1992.

[BDD+92] G. v. Bochmann, A. Debaque, R. Dssouli, A. Jaoua, R. Keller, N. Rico, and F. Saba. A method for architectural modelling and dynamic analysis of information systems and business processes. Technical Report CRIM 92/12/10, Centre de recherche informatique de Montréal (CRIM), Montreal, December 1992.

[Ber90] Lucy Berlin. When Objects Collide: experiences with reusing multiple class hierarchies. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 181–193, Ottawa, Canada, October 1990.

[Bis92] Walter R. Bischofberger. Sniff - a pragmatic approach to a C++ programming environment. In *Usenix C++ Conference*, Portland, OR, August 1992.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1991.

[BR87] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41–49, March 1987.

[Bro87] Frederick P. Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):1–14, April 1987.

[Coa92] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, September 1992.

[Coo87] Steve Cook. Panel P2: Varieties of inheritance. In *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications, Addendum to the Proceedings*, pages 35–40, Florida, USA, October 1987. Association for Computing Machinery. Panel Discussion.

[Deu83] L. P. Deutsch. Reusability in the smalltalk-80 programming system. In *Workshop on Reusability in Programming*, pages 57–71, (Newport, R. I. Sept), 1983. ITT Programming, Stratford, Conn.

[DK76] F.L. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[EG92] Thomas Eggenschwiler and Erich Gamma. Et++swapsmanager : Using object technology in the financial engineering domain. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, Vancouver, B.C., October 1992.

[ES92] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, 1992.

[FBPD+91] W.B. Frakes, T.J. Biggerstaff, R. Prieto-Diaz, K. Matsumura, and W.Schaefer. Panel 1: Is Software Reuse Delivering? In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 52–54, Austin, Texas, May 1991.

[Fra92] Ulrich Frank. Designing procedures within an object-oriented enterprise model. In *Proceedings of the Third International Working Conference on Dynamic Modelling of Information Systems*, pages 365–385, Noordwijkerhout, The Netherlands, June 1992.

[Fre87] Peter Freeman. A perspective on reusability. In *Tutorial Software Reusability*, pages 2–8. IEEE Computer Society Press, 1987.

[Gam91] Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge*. PhD thesis, University of Zurich, September 1991.

[Gar83] D. Garlan. The role of formal reusable frameworks. In *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, pages 57–71, (Napa, Calif.), May 1983. ACM Press, NewYork.

[Gea86] Eric Gullichsen and et al. The PlaneText Book. *Technical Report STP 333-86, MCC, Austin, Texas*, 1986.

[GHJV93a] E. Gamma, R. Helm, R Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object oriented design. In *European Conference on Object-Oriented Programming*, Kaiserlauten, Germany, July 1993.

[GHJV93b] E. Gamma, R. Helm, R Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object oriented design - A Catalog of Object Oriented Design Patterns. *Technical Report in Preparation, IBM Research Division*, 1993.

[Gos89] Sanjiv Gossain. *Object-Oriented Development and Reuse*. PhD thesis, University of Essex, UK, September 1989.

[GTC+90] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz, and Xavier Pintado. Class management for software communities. *Communications of the ACM*, 33(9):90–103, September 1990.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 169–180, Ottawa, Canada, October 1990. Association for Computing Machinery.

[HM84] E. Horowitz and J.B. Munson. An expansive view of reusable software. *IEEE Transactions on Software Engineering*, SE-10(5):477–487, September 1984.

[Hol92] Ian M. Holland. Specifying reusable components using Contracts. In *Proceedings of Tools 92*, pages 287–308, Paris, France, 1992.

[JBB+92] A. Jaoua, J.M Beaulieu, N. Belkiter, A.-C. Debaque, J. Desharnais, R Lelouche, T Monkam, and M. Reging. Rectangular decomposition of object-oriented software architectures. Technical report, Laval University, Québec, June 1992.

[JF88] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 4(2):22–35, June/July 1988.

[Joh92] Ralph Johnson Documenting frameworks using patterns. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 63- 76, Vancouver, B C., October 1992.

[JS89] Christina Jette and Reid Smith. Examples of reusability in an object-oriented programming environment In T. Biggerstaff and A. Perlis, editors, *Software Reusability*. Addison-Wesley, Reading, MA, 1989.

[KCTT91] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 268–218, Austin, TX, May 1991.

[KLO+93a] Rudolf K. Keller, Richard Lajoie, Marianne Ozkan, Fayez Saba, Xijin Shen, Tao Tao, and G. v. Bochmann. The Macrotec toolset for CASE-based business modelling. In *Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering*, pages 114–118, Singapore, July 1993.

[KLO+93b] Rudolf K. Keller, Richard Lajoie, Marianne Ozkan, Fayez Saba, Xijin Shen, Tao Tao, and G. v. Bochmann. User interface aspects in the Macrotec toolset for business modelling and simulation. In *Proceedings of the Fifth International Conference on Human-Computer Interaction (Poster Sessions: Abridged Proceedings)*, page 253, Orlando, FL, August 1993.

[KLS93] Rudolf K. Keller, Richard Lajoie, and Xijin Shen. Gxf+ file specification. Technical report, Centre de recherche informatique de Montréal (CRIM), Montreal, January 1993.

[KOR92] Rudolf K. Keller, Marianne Ozkan, and Nathalie Rico. Comparaison fonctionelle des outils de simulation. Technical report, Centre de recherche informatique de Montréal (CRIM), Montreal, July 1992.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[LCV87] Mark A. Linton, Paul R. Calder, and John M. Vlissides. Interviews: A C++ graphical interface toolkit. In *Proceedings of the USENIX C++ Workshop*, Sante Fe, NM, November 1987. Appeared as *The Design and Implementation of Interviews* and this is a later version.

[LHKS91] John A. Lewis, Sallie M. Henry, Dennis G. Kafura, and Robert S. Schulman. An empirical study of the object-oriented paradigm and software reuse. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 184–196, Phoenix, AZ, October 1991.

[Lip92] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, second edition, 1992.

[McG92] John D. McGregor. *Object-Oriented software development engineering software for reuse*. SAMS, 1992.

[MEN92] Alberto O. Mendelzon, Frank Ch. Eigler, and Emmanuel G. Noik. GXF: A graph exchange format. Technical report, University of Toronto, Toronto, July 1992.

[Mey87] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(3):50–64, March 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

[Mey91] Scott Meyers. Difficulties in integrating multiview development systems. *IEEE Software*, 8(1):49–57, January 1991.

[OBHS86] Tim O'Shea, Kent Beck, Dan Halbert, and Kurt J. Schmucker. Panel on: The learnability of object-oriented programming systems. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 502–504, Portland, Oregon, September 1986. Association for Computing Machinery. Panel Discussion.

[OH92] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 25–40, Vancouver, B.C., October 1992.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 145-160, September 1990.

[PD91a] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88-97, May 1991

[PD91b] Rubén Prieto-Díaz. Making Software Reuse Work: An implementation model. *ACM SIGSOFT Software Engineering Notes*, 16(3):61-68, July 1991.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[RGP88] C. V. Ramamoorthy, Vijay Garg, and Atul Prakash. Support for reusability in genesis. *IEEE Transactions on Software Engineering*, 14(8):1145-1153, August 1988.

[ROL90] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[Sch86] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh*. Hayden Book company, Hasbrouck Heights, NJ, 1986.

[SD91] M.E. Scharenberg and H.E. Dunsmore. Evolution of classes and objects during object-oriented design and programming. *Journal of Object-Oriented Programming*, pages 30–34, January 1991.

[SGME92] Bran Selic, Garth Gullekson, Jim McGee, and Ian Engelberg. ROOM: An object-oriented methodology for developing real-time systems. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, pages 230-240, Montreal, Canada, July 1992.

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10-20, May 1988.

[TMWH92] K. S. Trivedi, J. K. Muppula, S. P. Woolet, and B.R. Haverkort. Composite performance and dependability analysis. *Performance Evaluation*, 14(3-4):197-215, 1992.

[VL89] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the Second Annual Symposium on User Interface Software and Technology*, pages 158–167, Williamsburg, VA, November 1989. Association for Computing Machinery.

[Vli90] John M. Vlissides. *Generalized Graphical Object Editing*. PhD thesis, Stanford University, June 1990.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.

[WBVC+90] Allen Wirfs-Brock, John Vlissides, Ward Cunningham, Ralph Johnson, and Lonnie Bollette. Designing reusable designs: Experiences designing object-oriented frameworks. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications, Addendum to the Proceedings*, pages 19–24, Ottawa, Canada, October 1990. Association for Computing Machinery. Panel Discussion.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

[WES87] Scott N. Woodfield, David W. Embley, and Del T. Scott. Can programmers reuse software? *IEEE Software*, 4(4):52–59, July 1987.

[WGM88] André Weinand, Erich Gamma, and Rudolf Marty. ET++ — an object oriented application framework in C++. In *Proceedings of OOPSLA'88*, pages 45–57, San Diego, California, October 1988.

[WGM89] André Weinand, Erich Gamma, and Rudolf Marty. Design and implementation of ET++, a seamless object-oriented application framework *Structured Programming*, 10(2):63–87, April-June 1989.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, New York, 1985.

[WMH93] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object oriented software. *IEEE Software*, 4(2):75–80, January 1993.

[WT91] Richard C. Waters and Yang Meng Tan. Toward a Design Apprentice: Supporting reuse and evolution in software design. *ACM SIGSOFT Software Engineering Notes*, 16(2):33–44, April 1991.