to appear in Proceedings of the 62nd Congress of the Association Canadienne Française pour l'Avancement des Sciences (ACFAS), Montreal, Canada, May 1994. Colloquium on Object Orientation in Databases and Software Engineering.

Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert

Richard Lajoie*   Rudolf K. Keller**

Centre de recherche informatique de Montréal (CRIM)
1801, McGill College, Suite 800, Montréal, PQ H3A 2N4, Canada
{larick, keller}@crim.ca

Abstract

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. Object-oriented frameworks further design-level reuse, in that they allow for reusing the abstract design of an entire application, modelling each major component with an abstract class. Yet, application design based on frameworks remains a difficult endeavor, and a comprehensive approach to represent and document designs based on frameworks is still missing. In our work, we have developed a multi-layered model for framework reuse which comprises reuse objects at different levels of abstraction, most notably, micro-architectures. We have adopted, refined, and integrated novel techniques for the representation and documentation of micro-architectures and frameworks, namely, design patterns, contracts, and motifs. We believe that our approach is a valuable step towards better exploiting the reuse potential of frameworks.

Keywords: Object-oriented, framework, design, reuse level, micro-architecture, design pattern, contract, motif.

1 Introduction

Many computer scientists see software reuse as a powerful means of improving the practice of software engineering. Brooks, for instance, 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 any challenges. According to Freeman, the state-of-the-practice of reuse is embarrassing [Fre87]. The present situation, and concern amongst researchers, is that software reuse is not living up to its original expectations [Kru92].

Object-oriented approaches have been touted as means for significantly improving software quality and the underlying development effort. Particularly, object-oriented frameworks are believed to facilitate sound software designs which lend themselves to reuse [JF88]. Yet, according to Wirfs-Brock and Johnson [WBJ90], several major research problems remain to be solved, including design and assessment of frameworks, application design based on frameworks, and adequate design notations for frameworks. Furthermore, the framework approach needs large-scale validation in practice.

Over the past two years, we have been involved in the development of Macrotec, a prototype environment for business modelling and reengineering [KLO+93, KOS94], which we have engineered with ET++ [WGM89], an object-oriented framework. When using the framework, we faced issues similar to the ones raised by Wirfs-Brock and Johnson. Specifically, we were confronted with the problem of understanding the preexisting software and with the challenge of designing and documenting our software for future reuse.

As part of the Macrotec work, we have developed a new approach to address these issues [Laj93]. The major contributions of our research are:

- the introduction and description of different levels of reuse in frameworks together with corresponding design components;
- the identification of framework micro-architectures [GHJV93a] as key components for reusable designs;
- the refinement of existing techniques for the representation and documentation of micro-architectures and frameworks;
- the integration of these techniques into a uniform model.

In this paper, we first review frameworks and introduce a multi-layered model for their reuse. Then, we detail the role of micro-architectures in the design and reuse of

* Author's current address: Bell Northern Research, Verdun, Québec. This work is based on author's research carried out at CRIM and McGill University.

**Author carried out part of this research as an adjunct professor at McGill University.

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

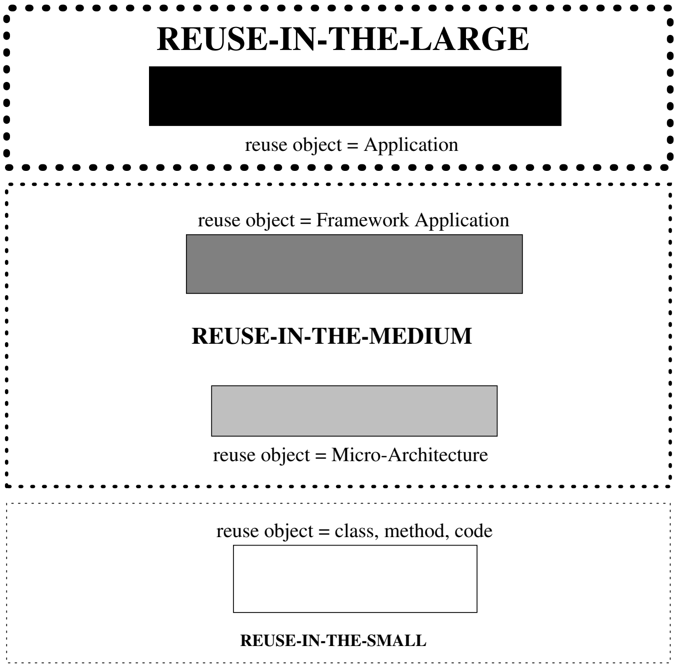

Figure 1: Different levels of reuse and abstraction in frameworks.

frameworks. Next, we present description techniques for both micro-architectures and frameworks, and discuss their application in concert. Finally, we draw some conclusions.

2 Frameworks and Levels of Reuse

In this section, we shall first discuss frameworks and their potential for reuse. Then, we shall introduce a multi-layered model for their reuse.

2.1 Frameworks and the Case for Reuse

An object-oriented framework, also known as *class framework* or simply *framework*, is a set of classes which provide a foundation for solutions to a set of problems. Frameworks are typically designed by experts in a particular domain and then used by non-experts. A framework allows for reusing the abstract design of an entire application, modelling each major component with an abstract class[1] [JF88].

Frameworks are *not* class libraries, they are in fact much more. A framework is the design of a set of classes that collaborate to carry out a set of responsibilities, an abstract design. 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.

A mature framework will typically have several concrete subclasses for many of its abstract classes, allowing for most applications to be developed by plugging together existing classes. Subclassing such a framework becomes a relatively trivial coding task due to the completeness of the abstract classes.

From a user's perspective, the major difference between a framework and a class library lies in the knowledge required to use them. The users of a class library need only understand the classes' external interfaces, but have to define themselves the overall structure of their applications. In contrast, users of frameworks have to understand the abstract design of the underlying framework as well as the internal structure of its classes, in order to adapt and extend them. Thus, in general, it is more difficult to learn using frameworks than class libraries. Also, frameworks carry the risk of being misused. However, their potential for reuse greatly exceeds that of class libraries.

1 Abstract classes are classes that are incompletely specified and designed to be subclassed rather than themselves instantiated.

2.2 Levels of Reuse in Frameworks

As Krueger [Kru92] points out, "there is a great diversity in the software engineering technologies that involve some form of software reuse." This applies also, on a smaller scale, to framework reuse.

A prerequisite for adequately describing and representing the different objects of reuse in frameworks, is their classification. As criterion for classification, we have chosen, as did Krueger and several others before [Kru92], the abstraction level of the different software artifacts involved.

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 and abstraction in frameworks (see Figure 1). The first, *reuse-in-the-small*, involves low-level reuse, i.e., the reuse of a *class*, *method*, and/or *code* fragment. For example, using a subclass that was derived from an abstract class is reuse-in-the-small. In fact, an abstract class is a small scale design that may be reused when designing small scale components. In contrast, the framework as a whole can be considered a large scale design.

The next level of reuse is *reuse-in-the-medium*, the reuse of classes and their interactions. We shall refer to this type of reuse as *micro-architecture* reuse.[2] As frameworks codify design knowledge of a particular domain, micro-architectures codify design knowledge in terms of the behavior of object collaborations. Micro-architectures comprise both the design and code of the classes involved and the interactions and control flow among them.

Reuse-in-the-medium involves an additional level of reuse, which is slightly higher in abstraction than micro-architectures and which comprises the interaction of micro-architectures. At this level, the objects of reuse are no longer classes, but are themselves micro-architectures. This is in fact the reuse of system architectures (*framework applications*) that have been instantiated from the underlying framework.

The highest level of reuse, *reuse-in-the-large*, is the reuse of *application* objects which are themselves independent systems and which are reused as they are, without being modified or extended in any way. It is the system which reuses them, the target system, which is responsible for adapting itself to the protocol requirements of the (pre-existing) applications. The applications may or may not have been developed with the same framework as the target system. They may have even been developed in a different programming language and/or with an approach that is not based on frameworks. Thus, the target system need not care about the design and implementation details of the applications, given that there is no intention of modifying or extending them.

In Figure 1, shading is used to represent the different abstraction levels of reuse objects and hence their varying degrees of encapsulation. Accordingly, the rectangle representing the objects of reuse-in-the-large, i.e., the off-the-shelf applications or *black box* objects, are depicted in black.

In the Macrotec development, our "case study," all three levels of reuse have been relevant. Yet, we feel that reuse-in-the-medium, especially the reuse of micro-architectures, shows greatest promise for successful reuse. Accordingly, we have emphasized this kind of reuse in our work.

3 Design and its Reuse in Frameworks

Application design in a framework should be oriented towards design-level reuse, in order to make best use of the framework. This poses particular design challenges, which we shall describe in the first part of this section. In the second part, we shall discuss the key role of micro-architectures in this kind of design.

3.1 The Challenges of Design

Design has long been considered hard and has been the cause of many difficulties in software engineering. Software design is difficult in part because designers, in most domains, work in the unknown, unguided by the history and traditions of earlier designs. This means that to date, design has mostly been "experience limited."

The design of flexible and reusable software is particularly hard. The ultimate goal of *design for reuse* is reusable software, which requires the design of general, extensible software components. Thus, designers are asked to predict the future uses of software and to incorporate the requirements for these possible future applications into the current design.

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. Furthermore, the design environment should support the creation of reusable software components.

Many designers using frameworks agree that working from a set of low-level components such as abstract classes and subframeworks provides many advantages to the design process. 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 as well as by the higher level, more abstract design of the underly-

2 The term *micro-architecture* is introduced by Gamma et al. in [GHJV93a].

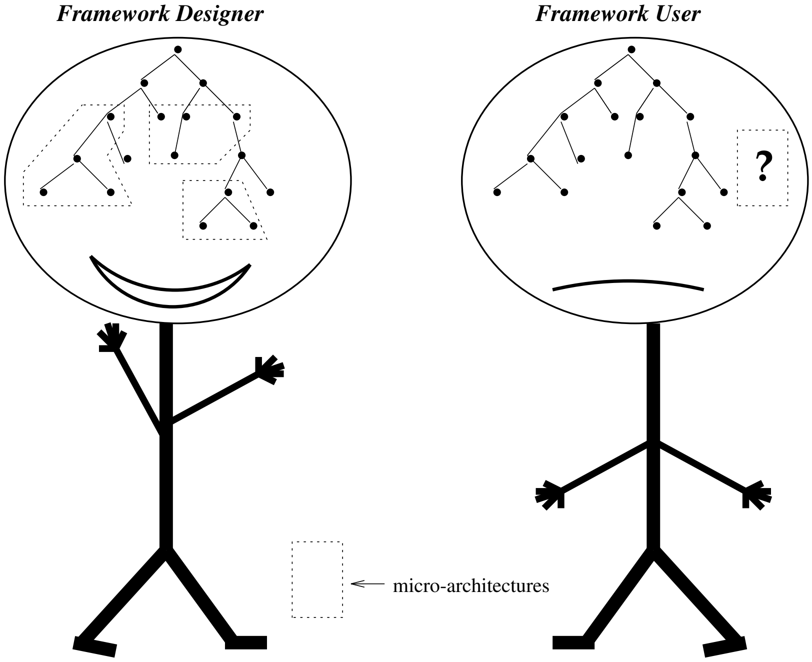

Figure 2: Conceptual views of frameworks: designer view (left) versus user view (right).

ing framework. We therefore suggest a design *guided* by the design of the underlying framework and possibly of its subframeworks and that a mixed *top-down/bottom-up* approach be taken.

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. Framework-based design reuse techniques are only now beginning to receive greater attention for their potential for reuse pay off [WBJ90].

3.2 The Role of Micro-Architectures

Clearly, frameworks provide a means for the reuse of 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 easily expressed. This makes frameworks harder to understand than, for instance, abstract classes.

As class frameworks codify design knowledge of a particular domain, micro-architectures, as introduced in the previous section, codify design knowledge in terms of the behavior of object collaborations. Micro-architectures are a way to abstract and to reuse design experience.

According to Gamma et al. [GHJV93a], there are object-oriented design structures that emerge repeatedly in the development of frameworks. These structures, micro-architectures, are of course known by the framework designers, but unfortunately by very few others. Figure 2 illustrates the situation: the informed framework designer (left) has a comprehensible, coarse-grain picture of the framework, whereas the novice framework user (right) is overwhelmed with the many, seemingly unrelated framework classes.

Providing designers and application developers with a set of micro-architectures specified by way of design patterns may be beneficial in several ways (cf. [GHJV93a]). Design patterns may:

- 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, the framework itself;
- provide targets for the refactoring of class hierarchies.

4 Describing Micro-Architectures

In this section, we shall discuss design patterns, as introduced by Gamma et al. in [Gam92, GHJV93a], and contracts [HHG90], two techniques for describing micro-architectures at a very high level and an intermediate level respectively. Unlike frameworks, which are expressed in a

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 3: Design pattern template (based on [GHJV93a]).

programming language, these techniques depend on special purpose notations.

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 architectural design or style. Alexander, an architect, 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,[3] each framework and its derived applications are ultimately implemented out of micro-architec-

3 A detailed discussion of the relation between Alexander’s patterns and design patterns is given in [Lea94].

tures, they are the atoms and molecules from which a system is designed and built. Design patterns, as presented by Gamma et al. in [GHJV93a] (see also [Coa92]), are a new way to identify and name micro-architectures. They are a mechanism for expressing how components interrelate as well as a high-level representation technique for properly capturing and expressing design experience and intent to ultimately facilitate design reuse. Design patterns may be described in an informal, template-based manner (see Figure 3).

A designer's familiarization with a set of design patterns (micro-architectures) is the starting point in design reuse. We, in developing the Macrotec prototype from the ET++ framework [WGM89], 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.

According to our experience, major ways to make use of design patterns include:

- as a guiding *blueprint* to introduce a new design;
- as a *reference* to ensure that hierarchy restructuring and/or class redesign do not affect behavioral collaborations;
- as a means to *understand* the underlying framework and/or application(s) developed from it.

In using design patterns for the purpose of understanding and reference, pattern identification may be a major problem. We therefore suggest that new implementations of a design include an identification of the guiding pattern in the component classes. This would allow one to easily identify the correct pattern. Moreover, 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 names as given in the design 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.

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. For instance, abstract design patterns allow designers to edit, combine, and remove 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 below), help designers understand some of the intended details behind the design. For this reason, we have modified the design pattern template originally proposed in [GHJV93a] to include the section *Contract Examples* (see Figure 3). This section is intended to make explicit references to different contracts developed from the design pattern in question.

4.2 Contracts

Design patterns describe framework design at a very high level. To ease the derivation of concrete designs from them, an intermediate representation is required. This intermediate representation applies best to *behavioral* design patterns, the patterns that are typically found at the core of most application designs. Often these intermediate representations are called contracts [HHG90, Hol92]. A contract is 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 no consensus yet as to contract formalisms and their level of abstraction. Attempts to use special-purpose programming languages have not been successful to date [Hol92]. Meyer has introduced contracts in the *Eiffel* language [Mey92], however, at a rather low level of abstraction. A recent approach by Wu and Barbeau attempts to apply them to requirements engineering and to transform them into colored Petri nets [WB94].

Most approaches, including our own, are based on some informal notation, typically pseudo code. The contract template presented in Figure 4 illustrates our contract description technique. Each contract begins with cross-references to the pertinent design pattern and/or motif(s) (described next section). 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.

Contracts are an intermediate representation between a micro-architecture and its corresponding design pattern. A contract may therefore be an intermediate development requirement when applying a design pattern to the creation of a new micro-architecture. Conversely, it is expedient to develop a contract as an intermediate stage in

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 4: Contract template.

the derivation of a behavioral design pattern from a micro-architecture.

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. We 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 refactoring affects 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.

5 Describing Frameworks with Motifs

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.

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 5: Motif template.

Novice users are interested in general, high-level aspects of the framework, before dealing with the design details of micro-architectures. We propose the adoption 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 the previous sections. We have introduced the term motif[4] in order to avoid confusion with design patterns. Below, we shall provide an overview of motifs, for details refer to [Laj93].

Each motif describes a situation which must be replicated in order to use the framework. All motifs have the same format (see Figure 5). 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., Macrotec. 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 may describe

- the underlying class 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. Macrotec. 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.

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, may be able to maintain the design intent during modifications ensuring that behavioral compositions are properly maintained.

6 Design Patterns, Contracts, and Motifs in Concert

The documentation of a framework should encompass:

- the purpose of the underlying framework;
- 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.

Motifs complement 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

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

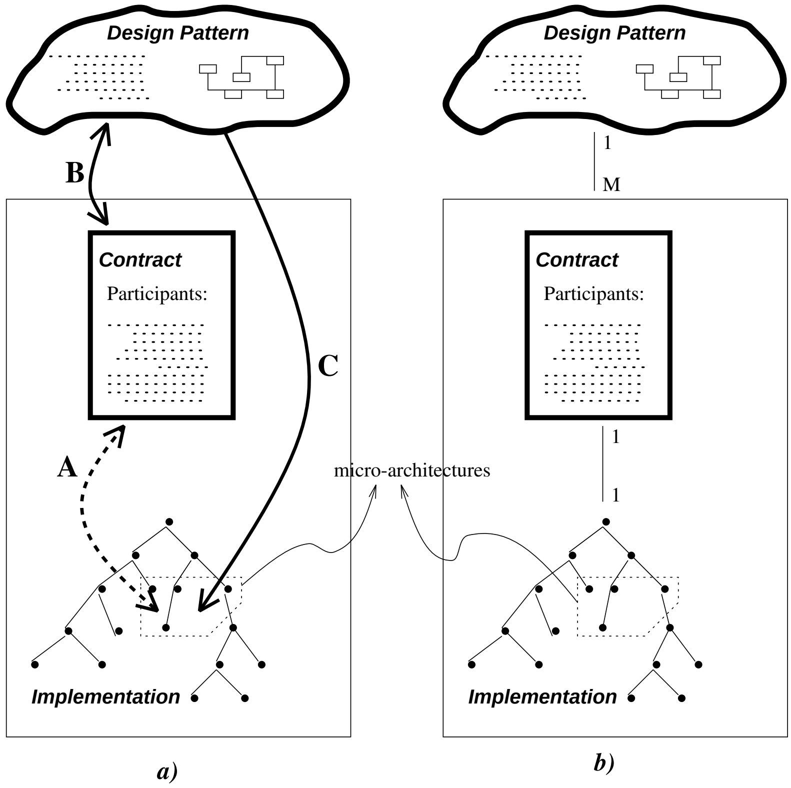

Figure 6: Descriptions of micro-architectures: development paths (left) and relationships (right).

any and all aspects of a framework that potentially require adaptation or need clarification.

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 (a class and/or method may participate in more than one micro-architecture).

Having classes reference their corresponding documents prevents implementors from inadvertently changing the responsibilities of a class that others may be gravely depending on.

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.

Section 4 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 6 (right), a design pattern may be applied to many different contract/micro-architecture implementations, and we have a one to many (1:M) relationship. 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 6 (left) depicts possible development paths for contracts and design patterns. 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 (see Figure 3);
- Through an intermediate contract (path B-A). 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 Figure 6 (left).

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 Figure 6 (left). 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 4.1).

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.

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]."

7 Conclusions

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 successfully 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 class framework's learning curve and consequently increase their 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.

We are not claiming, with the contents of this paper, 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."

For lack of space, we have not provided examples in this paper. A rich collection of design patterns and examples can be found in [GHJV93b]; our work is further illustrated in [Laj93].

Just as the Inuit 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 of software engineers as snow is in the life of the Inuit.

Acknowledgement

We appreciate the contributions of Marianne Ozkan, Xijin Shen, and Tao Tao, our colleagues in the Macrotec development team. Our thanks also go to Erich Gamma and André Weinand for providing us with the ET++ framework, to Richard Helm for teaching us about design patterns, and to Walter Bischofberger and Bruno Schäffer, the engineers of the *Sniff*+visual programming environment for C++.

References

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

[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.

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

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

[Gam92] Erich Gamma. *Object-oriented Software Development exemplified with ET++: Design Patterns, Class Library, and Tools*. Springer-Verlag, Berlin, Germany, 1992. Book version of PhD thesis, University of Zürich, Switzerland, 1991; in German.

[GHJV93a] E. Gamma, R. Helm, R Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object oriented design. In *Proceedings of the 7th EuropeanConference on Object-Oriented Programming (ECOOP '93)*, Kaiserlauten, Germany, July 1993.

[GHJV93b] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. A catalog of object-oriented design patterns. Technical report, Taligent Inc., Cupertino, CA, 1993. preprint.

[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.

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

[Hol92] Ian M. Holland. Specifying reusable components using Contracts. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP '92)*, pages 287–308, 1992. em LNCS/ 615.

[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.

[KLO+93] 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.

[KOS94] Rudolf K. Keller, Marianne Ozkan, and Xijin Shen. Towards comprehensive support for the dynamic analysis of Petri net based models. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, Zaragoza, Spain, June 1994. to appear.

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

[Laj93] Richard Lajoie. Using, reusing and describing object-oriented frameworks. Master's thesis, McGill University, Montreal, PQ, Canada, July 1993.

[Lea94] Doug Lea. Christopher Alexander: an introduction for object-oriented designers. *ACM SIGSOFT Software Engineering Notes*, 19(1):39–46, January 1994.

[Mey92] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[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.

[WB94] Qun Wu and Michel Barbeau. Verification of object-oriented specifications. Technical report, Université de Sherbrooke, Sherbrooke, Québec, Canada, January 1994.

[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. Panel Discussion.

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

[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.