Making Design Patterns explicit in FACE, a Framework Adaptive Composition Environment

Theo Dirk Meijler, Robert Engel

Software Composition Group, University of Berne[1]

Washington University, Computer Science Department[2]

Abstract. Creating applications using object-oriented frameworks is done at a relatively low abstraction level, leaving a large gap with the high abstraction level of a design. This makes the use of a framework difficult, and allows design and realization to diverge. Design patterns are more specific elements of design, and thus reduce this gap. We even bridge this gap by making design patterns and the classes that play a role within them into special purpose software components. System realization becomes a matter of composing special purpose class-components. We also introduce a system, FACE, which supports the visual composition of such specifications.

1 Introduction

When comparing the development of applications using frameworks [6] to the development of applications using libraries or from scratch, using a framework is —after a learning period— significantly less labour intensive [19]. Thus frameworks have a large commercial value.

Still, the use and evolution of a framework has many pitfalls. Many problems are due to the large gap between realizing a system and designing a system. Realizing a system using a framework is done at a low abstraction level. It often requires the creation of subclasses, which requires a profound knowledge of the implementation of the superclass [9]. Understanding is also needed of how the classes cooperate, thus how the implementation corresponds to the overall design. On the other hand, while a high-level simple design description to understand a system is clearly needed, currently used design notations, such as e.g. a class-diagram [3], are too generic to convey the intent of the design. The gap between the relative low abstraction level of the realization and the high abstraction level of the design is even such that the realization and the design are causally independent. It is quite natural to adapt the realization without adapting

1. *Author's address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.3314. *Fax:* +41 (31) 631.3965.
E-mail: meijler@iam.unibe.ch. *WWW:* http://iamwww.unibe.ch/~meijler.

2. Department of Computer Science, Washington University, Campus Box 1045, St. Louis, MO 63130, USA. *Tel:* +1 (314) 935 8501.
E-mail: engel@cs.wustl.edu. WWW: http://siesta.cs.wustl.edu/~engel.

the design. The realization may thus introduce classes that have not been given by the design, but even worse, it may “break” the design.

Design patterns [4] do not only provide solutions for building flexible systems. They are also important in reducing the gap between implementation and design. They provide more “special purpose” design elements, and make it easier to understand, at implementation level, the connection with the design. Still, the principal problem of bridging the gap between design and realization remains.

We present here an approach to compositional software specification that intends to bridge the gap between design and implementation. The specifications are at such a high level of abstraction that they are close to a design. On the other hand, since the specification consist of composition of special purpose components, it provides enough information (apart from certain details such as the implementation of specific methods) to directly represent the realization itself. Thus, design and realization no longer need to be separate[1].

That systems may be specified at a high level of abstraction in terms of compositions of domain specific *objects* is becoming good practice as several of the well-known design patterns show [4]. The main contribution of this approach is that such object-level system composition can be complemented by composition of domain specific “class-components”, which is at a similar high level of abstraction, and close to a “traditional” design (in e.g, the Unified Method [3]), but is also close to a realization. Composition of special purpose class-components is especially appropriate in the context of design patterns: a specific use of the design pattern may be seen as a composition of special purpose class-components, where for each of the roles that classes play in the design pattern there is a special purpose class-component. This approach circumvents the need for subclassing, since system realization is only done by composing and parameterization of class-components[2].

In the context of this overall approach, we introduce in this paper the Framework Adaptive Composition Environment “FACE”. The main contribution of FACE is that it is really a “framework adaptive” visual composition environment: It provides visual support for correctly building compositions and it can be adapted to the kinds of components (objects and/or classes) that are needed in a certain domain and to the rules there are for composing them —Note that we thus see a domain specific set of objects and class components, together with the composition rules as a (“component-oriented”) “framework”—. Salient point here is, that FACE is itself an application of the ideas on class-composition mentioned above. The composition rules are themselves embodied as a “descriptive” composition of special-purpose class-components, so that the visual composition system is itself specified by this descriptive composition. This thus shows the power of the approach to class-composition. Moreover, it allows to let visual composition, and the running of applications be specified by means of visual composition to form a seamless whole.

1. We note that, as described in the CACM of february 1996 [14], it is already possible in the area of manufacturing to have a similar direct connection between the “design” of a product and the realization of the corresponding manufacturing process. This may serve as an important model!

2. Note that this kind of parameterization goes much further than the parameterization used in template classes as in C++ [10]

In this paper we illustrate the approach. We focus on the ideas for class-level composition. We shall use the state design pattern as our example. In the further discussion a specific terminology is used (when necessary) to distinguish between the different contexts in which the word design pattern may occur.

- The "pattern description", as it is given in (e.g.) the "Design Patterns" book. A short description of the State pattern will be given in section 2.
- The "instantiated pattern", which is a class-component structure that makes the use of the pattern explict. In section 3 we show how the State pattern may be instantiated. Such a composition contains more information than a class diagram: In section 4 we describe what such a composition means at run-time.
- The "instantiated pattern instantiation", which corresponds to the run-time objects that are instances of the class-components in the instantiated pattern. In section 5 we shall show how the class-composition get its meaning for the behaviour of the run-time objects
- The "applied pattern" which is a set of rules that describes the possible evolution of the class structure, and the kind of class-components that may occur. This will, in general, form part of a complete set of rules for class-structure evolution of the framework. We shall however not discuss combining patterns in this paper. In section 6 we show how composition rules that describe the applied pattern are embodied as components.

In section 7 we describe how the composition rules are used to drive the visual composition environment. In section 8 we present related work, in section 9 we conclude. We stress that all discussions that are specific for the state pattern illustrate a general approach.

2 Short Description of the State Design Pattern

The main example in this paper will be the State design pattern. We refer to [4] for details. The state design pattern is used when an object has to exhibit state dependent behaviour, that is, dependent on the state the object reacts differently to the same requests for operation execution. The solution the pattern offers is illustrated in figure 1 for an object p that has to implement the

"TCP" network protocol. The object delegates requests that have to be handled state dependently to another object (the state object). It changes state by exchanging that object.

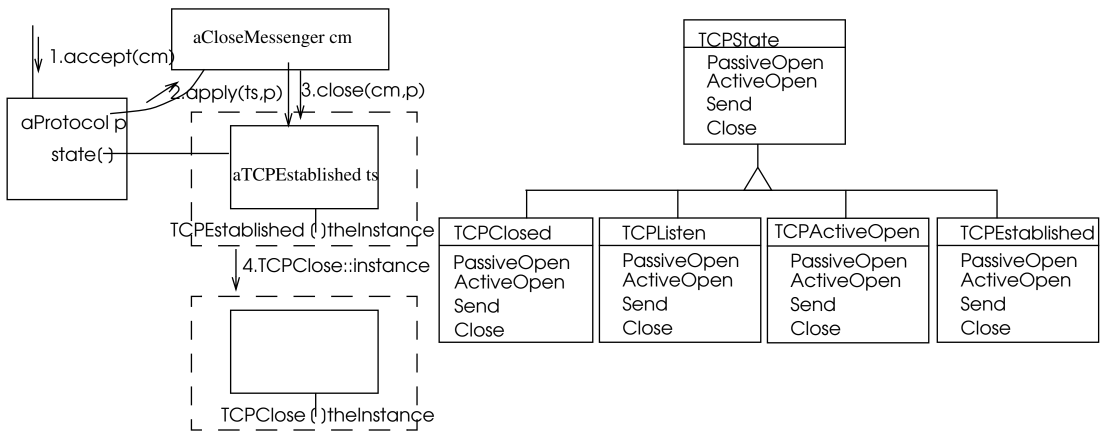

Figure 1. Solution to implementing the state design pattern as used by Hueni et.al.[5]. Left: An object p refers in its property "state" to one of the state objects. Each of these state objects is a "singleton" instance of the classes given at the right side. The figure shows a typical sequence of messages. The class hierarchy at the right side shows that each state object is an instance of the abstract class TCPState, which defines the basic messages that can be sent in each state. Each of the subclasses specifically defines the behaviour of the object for each of these states for each of these commands.A fully implemented protocol needs more states. Only subclasses are shown for states needed to establish a connection and for the closed state. Note that we shall call "messengers" as used in [5] "requests" in the text.

The TCP network protocol implementation that is an example of the use of the state pattern is taken from [5]. The solution is somewhat different from the standard solution [4] in that, instead of the context object (here called a "protocol" object) directly reacting to messages, and delegating these to its state object, there will be a request object[1] that represents the operation to be executed. The protocol handles the request by giving it a reference to the state object and sending it the message "apply". The request then sends the corresponding message to the state object. In this way the protocol class can be generally applied in any use of the pattern, independent of what kind of operation requests may need to be handled.

3 An Instantiation of the State Design Pattern

Figure 2 illustrates how a particular instantiation of the state design pattern may be specified as a composition of class-components. In contrast to the diagram in figure 1, this is not a figure that merely represents a design: It is a direct representation of the class-component structure, and thus (with details left out) of a *realization*. FACE may present such a structure in a similar way

1. In [5] this is called a messenger object. We follow however the general terminology in [4].

to the user. We shall only discuss what kind of components are shown, in section 4 we will discuss the meaning of such a composition.

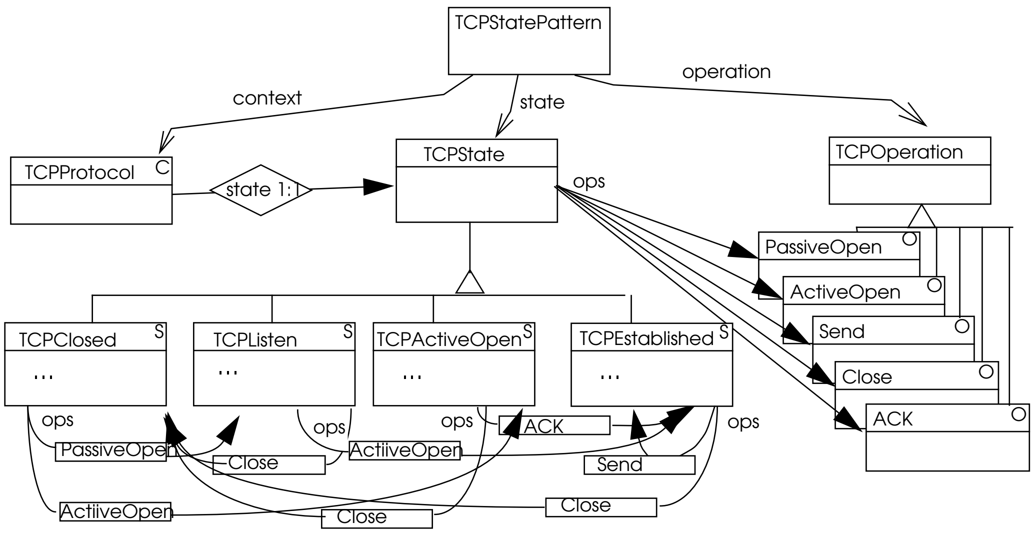

Figure 2. Realization of the TCP Protocol as a composition of state pattern specific special purpose class-components. This is thus an explicitly represented instantiation of the state design pattern. All rectangles in the figure are special purpose components. The rectangles with a line in them are class-components. The link between the class-components labelled with “S” and the “TCPState” class-component is a subtyping relationship, as represented by the open triangle. The diamond represents an association descriptor. Some of the links between components are labelled: e.g., “ops” represents the set of operations of a class-component. See the text for further information.

Different kinds of components in the figure.

1. A container component is shown (name “TCPPattern”) that represents through its type (the “MetaStatePattern”, see section 6) the fact *that* the state design pattern is being instantiated. It refers via special purpose slots for each of the “roles” in the pattern to all the class-components that play that specific role.
2. Class-components are shown represented by rectangles with a line as is normal in (e.g.,) the unified method [3]. Although class-components are like classes, in that they may have instances at run-time, there are two main reasons why we use this term instead of classes. Firstly, they are domain specific components used to compositionally specify a system. In the figure the specific purpose (type) of a class-component is represented by the character in the top right corner. Secondly, the class-components are “black-box”. Only their linking and parameterization is “visible”. They do not show any information about the implementation of the instances as this is usual for classes. Specialization of a class-component is thus done by means of parameterization and linking (composing) only.

   There are special purpose class-components for each of the different roles that classes may play in the design pattern: In this example, as corresponding to the terminology in [4], there are operation class-components (represented by an “O”) the instances of which represent requests (called messenger objects in [5]) that can be executed by the protocol object. There is one “context” class-component (represented by a “C”) here with the name TCPProtocol, the instance of which represents the protocol object which handles the request objects by delegating them. And there are state class-components (represented by the character “S”), instances of these handle the delegated request instances. All state class-components are subtypes of (as indicated by the open triangle) the abstract class-component here called “TCPState”. “TCPState” defines in its “ops” (short for operations) property what kind of requests any state object may handle.

For each of the different kinds of special purpose-components there are rules that describe its “composition/parameterization signature”, i.e., how it may be connected to other class-components, how it may be parameterized. See also section 6.

3. Also shown in the figure, represented as lozenges, is a so-called association descriptor. Association descriptors are generally used to specify a possible link between instances.
4. There are also other special purpose components. In the figure, each of the state class-components is linked via a bent arrow to a transition descriptor. The state-class component will be called the “transition starting point” of the transition descriptor. The transition descriptor has a name, which corresponds to the name of one of the operation class-components, and a transition target, which is another class-component.

4 Meaning of a composition.

The meaning of such a composition has two main aspects: Meaning for the run-time behaviour of an application and meaning for the allowed compositions of objects. See again Figure 2. we do not intend to give an explanation of the full figure.

a) Meaning for run-time behaviour.

How a composition determines the run-time behaviour is totally pattern specific, and depends on each component and its component-type. Generally valid is however the basic principle: For any special purpose class-component, given its role in the design pattern, much of the behaviour of its instances is fixed. The links to other components, and parameters fill in the unknown aspects of this behaviour.

For the instantiated state design pattern the transition descriptors play a major role in the specification of the behaviour. Given its transition starting point, its name that corresponds to a name of a operation class component, and its transition target, it specifies how an instance of the same-named operation class-component should be handled by an instance of the state-class that is the transition starting point. The handling of the request should result in the protocol object making a transition, by referring to the instance of the target state-class component. See also section 5. The transition descriptor will also contain other parameters (not shown in the figure) that give information about the behaviour when handling that request, e.g., whether the request should be copied and send on, deleted, or otherwise...

b) Meaning for possible compositions of objects.

As mentioned in the introduction, a complete system specification will consist of an object-composition (an instantiation of the intstantiated pattern) as well as of a class-composition. The object-composition describes the initial set of relevant run-time objects.

In general (!), an association descriptor that links two class-components describes a possible link in the object-composition between two instances of these class-components. Thus, in the example the association descriptor with the name “state” shows that in the object-composition the protocol object should be connected to a certain state object, representing the initial state of the protocol. Note that these kinds of association descriptors (although differently presented[1]) are also used in traditional designs. Here association descriptors only represent links that are of relevance for the specification through the object-composition while in a traditional design other links may be declared as well.

Of relevance is that the class-composition thus contains information regarding the rules for object composition, which is directly integrated with the rest of the composition.

5 Realizing the Run-Time Meaning of the Class-Composition.

From a point of view of programming languages, special purpose class-components are framework specific abstractions. There are two principal ways of realizing the semantics of such abstractions:

1. By mapping them to lower-level general abstractions; that is, through an extensible compiler
2. By using ideas from Metaobject Protocols [7] (MOP’s).

We use approach 2 as described in the following. Approach 1 is under research in our group as well [22], but will not be discussed in this paper.

In our approach, class-components are reified (explicitly represented) at run-time as objects as is usual in MOP’s. Thus all links and parameters of class-components are explicitly accessible at run-time as well. The instances of the class-components (which will be called “basic objects” to make a distinction) refer to their class-components. The basic objects implement the “generic” behaviour. When certain information is supposed to be given by the class-component (e.g., which transition to which other state should be made) the basic object will make an “upcall” to its class-component to query it about this information and adapt its behaviour accordingly. The class-component in its role of run-time accessible object thus implements behaviour to answer these queries.

1. We present an association descriptor as one separate lozenge, to show that there is one corresponding component in the class-composition.

We describe in further detail how the state class-components and their linking via transition descriptors determine the behaviour of the corresponding instances, see figure 3.

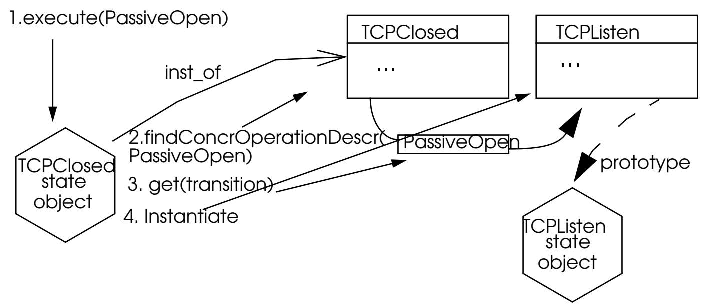

Figure 3. Object interaction for making a state object execute a transition diagram: The state object does an up-call to its class-component, in this case to "TCPClosed". "TCPClosed" finds the concrete operation descriptor for "PassiveOpen". The state object retrieves the information from this object, especially the reference to the next class-component, being "TCPListen". The state object asks it for instantiation. "TCPListen" returns its prototype. This instance is returned to the calling protocol object.

Assume that the TCP-Protocol object p refers through its state property to an instance of the state class-component "TCPClosed", it has to handle a request that is an instance of "PassiveOpen". The default behaviour of the protocol object is (as described in section 2), to send the message "apply" to the request, together with a reference to the state object. As a result (this is shown in figure 4) the request will send the message "execute" to the state object, with the name "PassiveOpen" as a parameter. The state object will query its class-component "TCPClosed" for the transition descriptor using the upcall message "findConcrOperationDescr". TCPClosed will return the transition descriptor. Next, the object will query the transition descriptor for the transition target. It will return a reference to the class-component "TCPListen". The state object will ask "TCPListen" for an instance ("TCPListen" has only one instance). This instance will be returned to the protocol object, that will change the value of its "state" property to refer to this other state object.

A short remark about implementation aspects: As mentioned before a class-component is not really a class, in the sense that it does not contain implementation specific information. The implementation of its instances will be specified elsewhere in an "underlying" programming language. The question is thus, how a class-component can create an instance. This is realized by applying the "prototype" design pattern [4]: Each class-component carries a prototype instance, that it may copy to create a new instance. Following the description of Hueni et.al. [5], state class-components will only have one instance: When the class-component is requested for an instance, it will always directly return its prototype instance.

6 Rules for describing class-compositions: Applying the pattern

Figure 3 illustrates how the applied state pattern may be represented as a composition of "meta-class-components" that embody the rules for correctly creating class-compositions following the pattern.

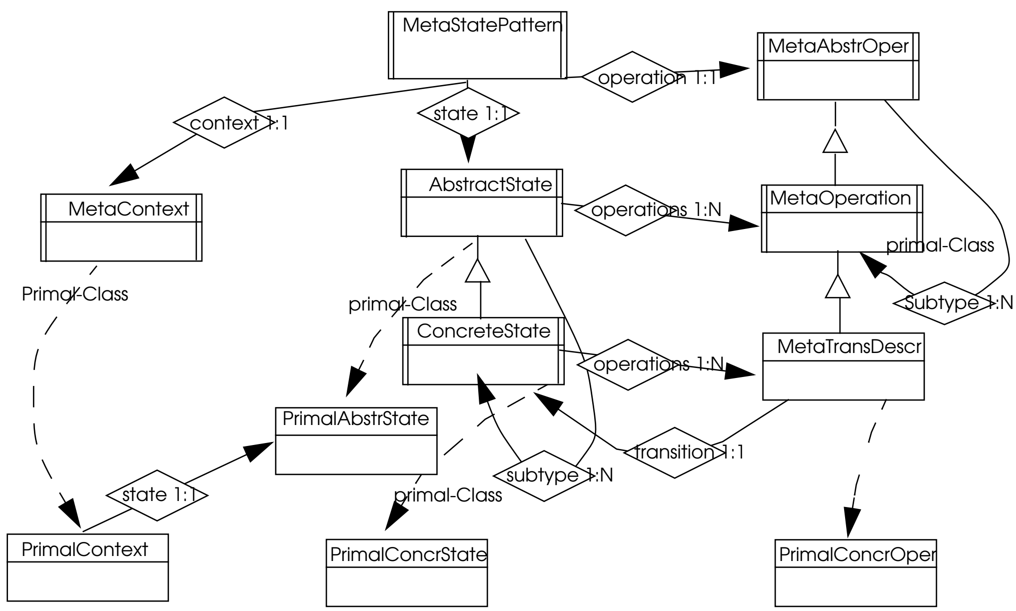

Figure 4. Meta-component structure for the state composition pattern. Metaclass-components are indicated with double lines at the sides. Association descriptors are again indicated by lozenges. See further the text and the explanation of Figure 2. Note that since "MetaTransDescr" is a subtype of "MetaOperation", transition descriptors, which are instances of MetaTransDescr are really operation descriptors.

The structural relationship between a metaclass-component and its instances is the same as between a class-component and its instances, see section 3, point 2. Thus, a meta-class component defines the existence and possible usage of a certain type of special purpose class-components. In the figure, for example, "ConcreteState" defines the type of special purpose state class-components; instances of which are indicated by "S" in figure 2. Note that a special type of abstract state class-components is defined: There is only one instance of this meta-class component in figure 2, namely, the supertype[1] of all concrete state class-components.

While class-components are used in two ways, namely, for specifying run-time behaviour and for specifying allowed compositions of their instances, meta-class components are mainly used for specifying allowed compositions of their instances. This is also done by means of association descriptors. For instance the "subtype" association descriptor between "Abstract-

1. We have not, and will not give details about subtyping in this paper. See [15] for details. Basically, a subtype may extend or specialize the compositional interface (as defined through association descriptors) as defined by the supertype, and it will extend or specialize the run-time interface in terms of operations and messages that can be handled.

State" and "ConcreteState" describes that an abstract state class-component has several instances of "ConcreteState" as subtype.

The figure shows that meta-class-components have "primal-classes" [15]. These primal classes serve a double role: As the prototype, and as the supertype of class-components that may be instantiated.

Primal-classes serve as the prototype that will be copied in order to create an instance of the meta-class-component. Thus, this primal-class component is an object that implements the run-time basic behaviour that all instances of the meta-class-component will have. This encompasses especially that behaviour that allows the class-components to be queried at run-time. Note that this is the same principle as already mentioned in section 5, where class-components have a prototype. The reason that prototypes of class-components have not been shown in figure 2 is that they are not used for any specification purposes. In their second role primal-classes *are* used for specification purposes.

The second role of primal-classes is to serve as a supertype: all instances of the meta-class-components will be a subtype of the primal-class. As in figure 3, where the primal-class of meta-protocol has an association descriptor to the primal-class of meta-state, this makes it possible in the meta-class-composition to enforce certain structural descriptions that are inherited in the class-composition. Thus, the meta-class composition, together with the primal-classes cannot only define the evolution of the class-composition, but also define default associations that have to exist in the class-composition.

7 Using reified rules to drive the visual composition environment.

The fact that rules for composition are embodied as class- or meta-class components and thus as components themselves is of major relevance in the FACE approach. This means in the first place that the creation of these rules is itself a composition process, and can be supported by the visual environment. Secondly, this means that the visual composition environment is itself an "application" of which the behaviour is specified by the composition that embodies the rules for correct composition. In the following we shall shortly describe how a class- or meta-class composition is used to drive visual composition.

We shall focus on how the visual environment checks the correctness of attempted connections. This will be done using the example: suppose that a user is developing a class-composition such as presented in figure 2. A new class component "TCPEstablished" has just been created, and an attempt is made to connect "TCPState" to "TCPEstablished" over the "subtypes" property. Also assume that when such an attempt is made that the corresponding association descriptor, namely the one for "subtypes" in figure 3, can be found (how this is done, and how the relationship between presentation and the "real" reified components is made is outside the scope of this paper). The attempted connection is checked by querying the association descriptor, similar as state object behaviour is adapted by querying the class-components: A message connectAllowed, with as a parameter a reference to the (meta) class-component of the component which is to be connected (in this case "ConcreteState") is sent to the association descriptor. The association descriptor matches the required type of the connected component (which is also "ConcreteState") with the one that is given as parameter. The type of the parameter may be the same, or a subtype of the required type. In this case the match succeeds directly, and the association descriptor returns "true" to indicate that the connection may be made. The user will be informed by means of a graphical clue whether or not a connection may be made.

8 Related work

There is a growing interest in building systems by means of composition. Commercially, so-called "component-ware" [27] has been successful. Visual Basic [18] (VB) is a well known example of a component-ware environment. One of the connections VB has with this work is that it allows for visual realization of a system. Components in VB are "instantiated" at run-time. However, they are not much more than "prototype" objects that are instantiated by copying. Thus class-level composition is not supported. Cooperations between components have to be explicitly implemented in the methods (event procedures) defined on each of them: There is no such thing as simply connecting components to specify their cooperation. Thus VB does not really specify object composition.

Vista [16] is a visual tool that does support object composition, and that can be adapted to the composition rules. Vista does not provide a way to give class-level compositions a run-time semantic. The composition rules in Vista are not expressed as compositions of components themselves as they are in our approach: So VISTA cannot support rule definition in the same way that it that it supports composition itself, as in FACE.

Composition at the level of classes (although not visually supported) is especially encountered in "generic" constructs, where classes can be parameterized with other classes, such as in the template classes in C++. The Standard Template Library STL [20] shows the power of applying this idea. Also the work of Batory et. al. [1][2] is based on parameterizing class-level components with class-level components. The work of McGee and Kramer on Darwin [13] represents another form of class-level composition: Links between components basically describe communication possibilities between the instances.

The model such as presented here may encompass both forms of composition due to the fact that the semantics of a connection is not fixed (can be defined at the meta-level). This paper describes links between classes (the transition descriptors) which fit in neither of these forms. These kind of links play an important role in the compositional instantiation of a design pattern.

Jiri Soukop [25] has described the possibility of making design patterns concrete. Certain similarities may be observed, especially in the strong use of genericity. Soukop's work does not seem to go as far in the use of genericity (parameterization of classes): A parameterization such as given by transition descriptors does not seem possible. We go further in bridging the gap between design and realization because of the visual support for composition that is embedded in our approach. Furthermore, we present a "total" approach to software composition which includes object composition. On the other hand, in Soukop's work more experience has been made, especially with respect to interlinking design patterns. His approach in letting a separate pattern class (comparable to our "container component") disentangle the dependencies between the classes seems interesting.

This work has roots in the area of open programming languages such as CLOS[7], and in Open Implementations[23]. Open programming languages such as CLOS reify their software components (classes), but the reification is not "black-box" the reification is too complex, and it is difficult to create “drastically” different components because of the intricate cooperation between all parts of the reified class.

The way in which we use reification and reflection is close to the model of Klas and Neuhold [8]. They focus on adaptive data models for databases systems, not so much on frameworks and application development. We also think that our model is simpler and thus easier to understand and use.

Steyaert [26] has described how the use of a meta-level interface could be applied to provide powerful configuration capabilities for user interface builders. Also, by the use of reflection new kinds of components could be described using the composition environment itself. However, the way in which Steyaert opens up the framework and corresponding visual composition seems to be restricted: He does not describe mechanisms for introducing new component cooperation forms in the framework.

The work of Lieberherr et. al.[11][12] on adaptive programming also shows how useful it is to have programs that adapt to the class-structure. They focus on adaptiveness of traversal operations. We think that our approach is broader. It may (but we still have to prove that) also cover traversal adaptiveness, although it is, of course, not as well “tuned” for that.

9 Conclusion

Making design patterns explicit is not a purpose in itself. The purpose is to bridge the gap between designing a system and implementing a system within the context of an arbitrary framework, so that he level of abstraction that is provided for realizing systems can be heightened, and that the problem of an implementation that is independent of the design is resolved. We have attempted to bridging this gap. Our approach may be viewed from two viewpoints: One could say that we have provided high-level domain specific abstractions (special purpose class-components) for realizing systems. One could also say that we have made design into realization by letting the designer work with domain specific components that can be put together. We have also briefly presented how a system may be realized that supports the developer in providing a graphical interface for building such compositions. The reason why such a system fits so easily in these ideas, is that such a system is itself specified using a composition: a composition that embodies the definition and rules of the components that may be put together using the system. At the same time we believe that this shows the power of this approach: How relative simple compositions may be used to specify complex systems, such as a graphical composition editor.

FACE will be finished to the extent that a demonstration of the example described will be possible by the publication of this paper. Our trust in the correctness of these ideas is based on a theoretical foundation in [15], preliminary experiments with implementation, and our first experiences of using these ideas to build the framework for supporting visual composition. Having a demonstration of an applied isolated design pattern is not sufficient to demonstrate the approach. In the future we hope to publish how patterns should be applied in combination by making encompassing meta-class structures. Important questions have to be asked how often special purpose class-components are really useful, or whether a specific implementation will still be needed for too many class-components. Also questions with respect to efficiency have to be asked: In the example we have totally disregarded efficiency; it is quite clear to us that the idea of using querying of the class-components to change the run-time behaviour “on the fly” is not efficient. Therefore, and for reasons of allowing reasoning about compositions further research on compiling special purpose abstractions is also of major relevance.

References

[1] Don Batory and Sean O’Malley, “The Design and Implementation of Hierarchical Software Systems With Reusable Components,” *ACM Transactions on Software Engr. and Methodology*, October 1992.

[2] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci and Marty Sirkin, “The GenVoca Model of Software-System Generators,” *IEEE Software*, Sept. 1994, pp. 89-94.

[3] Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development Version 0.8*, Rational Software Corporation, 1995.

[4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.

[5] Hermann Hueni, Ralph E. Johnson and Robert Engel, ‘‘A Framework for Network Protocol Software,’’ *Proceedings OOPSLA ’95, ACM SIGPLAN Notices*, to appear.

[6] Ralph E. Johnson and Brian Foote, ‘‘Designing Reusable Classes,’’ *Journal of Object-Oriented Programming*, vol. 1, no. 2, 1988, pp. 22-35.

[7] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press (Ed.), 1991.

[8] Wolfgang Klas, E.J. Neuhold and Michael Schrefl, ‘‘Metaclasses in VODAK and their Application in Database Integration,’’ *Arbeitpapiere der GMD*, no. 462, 1990 .

[9] John Lamping, ‘‘Typing the Specialization Interface,’’ *Proceedings OOPSLA ’93, ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 201-214.

[10] Simon Lippman, *The C++ Primer, Second Edition*, Addison-Wesley, 1991, (3).

[11] Christina V. Lopes, Karl J. Lieberherr, ‘‘AP/S++: Case-Study of a MOP for Purposes of Software Evolution,’’ *Proceedings Reflection ‘96,* to appear.

[12] Karl J. Lieberherr, Ignacio Silva-Lepe, Cun Xiao, ‘‘Adaptive object-oriented programming using graph-based customization,’’ *Commun of the ACM,* Vol 37, no. 5, May 1993, pp 94-101.

[13] Jeff Magee, Naranker Dulay and Jeffrey Kramer, ‘‘Structuring Parallel and Distributed Programs,’’ *Proceedings of the International Workshop on Configurable Distributed Systems*, London, March 1992.

[14] Marti Mäntylä, Dana Nau and Jami Shah, “Challenges in Feature-Based Manufacturing Research”, CACM, vol. 39, no. 2, February 1996, pp. 77-85

[15] Theo Dirk Meijler, ‘‘User-level Integration of Data and Operation Resources by means of a Self-descriptive Data Model,’’ Ph.D. Thesis, Erasmus University Rotterdam, Sept. 1993.

[16] Vicki de Mey, ‘‘Visual Composition of Software Applications,’’ *in [21],* pp. 275-303.

[17] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, 1992.

[18] Microsoft Corporation, *Visual Basic Programmer’s Guide*, 1993 .

[19] Simon Moser and Oscar Nierstrasz, “Measuring the Effects of Object-Oriented Frameworks on the Software Process,” submitted for publication, IAM, U. Berne, Dec. 1995.

[20] David R. Musser and Atul Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.

[21] Oscar Nierstrasz and Dennis Tsichritzis (Ed.), *Object-Oriented Software Composition*, Prentice Hall, 1995.

[22] Oscar Nierstrasz, “Research Topics in Software Composition,” *Proceedings, Langages et Modèles à Objets*, Nancy, Oct. 1995, pp. 193-204.

[23] Ramana Rao, ‘‘Implementational Reflection in Silica,’’ *Proceedings ECOOP ’91*, P. America (Ed.), LNCS 512, Springer-Verlag, Geneva, Switzerland, July 15-19, 1991, pp. 251-267.

[24] Alan Snyder, ‘‘Encapsulation and Inheritance in Object-Oriented Programming Languages,’’ *Proceedings OOPSLA ’86, ACM SIGPLAN Notices*, vol. 21, no. 11, Nov 1986, pp. 38-45.

[25] Jiri Soukop, “Implementing Patterns,” *Pattern Languages of Program Design,* Addison Wesley 1995, Chapter 20.

[26] Patrick Steyaert, K. De Hondt, S. Demeyer, N. Boyen and M. de Molder, ‘‘Reflective User Interface Builders,’’ *Proceedings Meta’95*, C. Zimmerman (Ed.), 1995 .

[27] James Udell, “Componentware,” *Byte*, vol. 19, no. 5, May 1994, pp. 46-56.