Making Design Patterns Explicit in FACE

A Framework Adaptive Composition Environment

Theo Dirk Meijler, Serge Demeyer, Robert Engel
Baan Labs (TDM)[1]
Software Composition Group, University of Berne (SD)[2]
Washington University, Computer Science Department (RE)[3]

Abstract. Tools incorporating design patterns combine the advantage of having a high-abstraction level of describing a system and the possibility of coupling these abstractions to some underlying implementation. Still, all current tools are based on generating source code in which the design patterns become implicit. After that, further extension and adaptation of the software is needed but this can no longer be supported at the same level of abstraction. This paper presents FACE, an environment based on an explicit representation of design patterns, sustaining an incremental development style without abandoning the higher-level design pattern abstraction. A visual composition tool for FACE has been developed in the Self programming language.

Keywords. Frameworks, Design Patterns, Software composition, Visual Composition, Reflection

1 Introduction

Design patterns [4] are gaining more and more attention as a technique for supporting the development and maintenance of object-oriented applications and frameworks. As a result of this success tools are being developed that support software engineers thinking and working at the level of design patterns as well as mapping these abstractions to an underlying implementation. Tools providing such a high level of abstraction for representing and dealing with patterns have indeed been described or have even reached the market (see the "Related Work" section).

Currently these tools generate source code and possibly documentation from the higher-level design pattern abstraction. Although such an approach provides the basic

---

1. Baan Labs, Groot Zonneoord, PO Box 250, NL-6710 BG Ede, The Netherlands. *Tel:* +31 (318) 69.6685. *E-mail*: meijler@research.baan.nl.

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

3. 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.

facilities for the initial development phases (i.e., rapid prototyping via code generation and library support), it fails to adequately support an incremental style of programming necessary to build mature frameworks and corresponding applications. The reason is that editing source code — almost inevitable in such an incremental process — breaks the implicit link between the higher abstraction level within the tools and the lower level implementation within the source code. This implies that once the source-code is changed all such tools become useless since changes to the higher-level representation code generation would override the "hand-made" changes. We call this problem the design-implementation gap.

Here we present FACE (Framework Adaptive Composition Environment), an approach that bridges this design-implementation gap by supporting incremental development using frameworks at the abstraction level of design patterns.

A framework is a software artifact that is specifically directed at enabling, through reuse, the easy development of applications in a certain domain. Current object-oriented frameworks, however, are still hard to use for application development, due to their use of subclassing as a specialization mechanism. The developer of the application is expected to have (almost) the same expertise as the person(s) who implemented framework. He has to understand the framework (the superclasses) almost as well as the framework developer does. Moreover, the application developer also uses the same language and tools as the framework developer.

FACE is, in contrast to object-oriented frameworks, an environment where a framework is used, and an application is built, on the basis of an approach that we call "modeling = programming." Application development is a task clearly distinct from framework development. Building an application is done by building a model which is described in terms of modeling primitives that the framework developer has defined. The purpose of this is to let the application developer work at a higher level of abstraction, circumventing the use of subclassing, and preventing, as much as possible the need for coding.

Design patterns are normally seen as micro architectures, reusable pieces of design, that link often occurring problems to a "best-practice" design, where a corresponding implementation is suggested but not enforced. At best a framework is made up of a combination of such micro architectures. In the modeling = programming approach of FACE the level of abstraction of design patterns coincides with the modeling abstractions of the application developer. Thus design patterns provide him with understandable modeling primitives with a hidden — framework specific as we shall see — implementation. Modeling will be a matter of defining the roles and relationships of classes in pattern-specific terms. For example, in the abstract factory pattern (see section 2), a factory class must be specialized by specifying its creation operations and specifying which creation operation instantiates which product class.

The model that must be composed by the application developer will be referred to as a *schema*. It is in general made using four kinds of modeling primitives: 1) classes and the role they play in the pattern, 2) operations and the role they play in the pattern, 3) relationships between classes and/or operations which may be pattern specific, and

4) parameters. Specifying which creation operation instantiates which product class, as mentioned before, is an example of a pattern-specific relationship.

When creating such a schema the application developer is said to *instantiate* the pattern. This goes as follows: A so-called *primal schema* consists of a basic set of abstract classes and their relationships capturing the essence of the micro architecture. This primal schema is cloned to form the *kernel* of the schema. Next, the kernel is extended by defining concrete classes with associated roles and operations, creating corresponding relationships and specifying necessary parameters. Since this must be done correctly, this must conform to a kind of "syntax" of available modeling primitives and how they may be combined. The modeling syntax is furthermore coupled to a semantics such that the run time behavior indeed corresponds to what was intended, e.g., with respect to which operation instantiates which product. We note explicitly that in order to allow for sufficient flexibility we may need to be able to parameterize a schema with source code.

We illustrate the FACE modeling = programming approach in this paper for individual patterns. Since — as in "normal" object-oriented frameworks — the full framework may encompass several design patterns the specialization of the framework goes accordingly: The developer copies a full primal schema to create the kernel schema and extends it. This also includes "non-pattern" specific relationships and classes. We will not further treat this in this paper.

The patterns that are presented here must be seen as "mini-frameworks." The reason being that it is generally known that there is no such thing as a standard implementation for a pattern; it is always fitted to the specific usage. Thus, the realization presented here cannot be simply copied to fit any framework. This is true for primal schema modeling primitives as well as semantics. The patterns have been chosen from the "Design patterns" book design pattern catalogue [4], each having a non-trivial class structure and belonging to a different pattern category, namely the "AbstractFactory" and the "State" pattern.

Although we focus in this paper on explaining and illustrating the programming = modeling concept of FACE through design patterns, and thus basically introduce a framework technology/ methodology, FACE allows (and needs) to let the modeling of the application developer be actively supported by a tool. We shall shortly discuss a prototype implementation of such a tool.

The rest of this paper is organized as follows: In section 2 we give a short description of the Abstract Factory design pattern. In section 5 the same is done for the State design pattern. In section 3 we show an example instantiation of the Abstract Factory design pattern in the FACE approach; in section 6 the same is done for the State pattern. In section 4 we present a *definition* of the Abstract Factory pattern which consists of the *meta-schema* describing the modeling syntax and a primal-schema. In section 7, this is done for the State pattern. In section 8, we show how a run-time meaning is given to schemas that instantiate the State design pattern. In section 9 we discuss how the syntax rules as embodied in the meta-schema are used to support the application developer in creating a correct schema. In section 10 we discuss related work, in section 11 we give a conclusion and discuss future work.

2 Short Description of the Abstract Factory Design Pattern

The first example in this paper will be the Abstract Factory design pattern. We refer to [4] for details. The Abstract Factory design pattern is used when client objects want to create certain objects (e.g., window objects, scrollbar objects) but the instantiation of these objects should not commit the client to choosing a specific implementation for these objects (e.g., either Presentation Manager Window or Motif Window). The basic idea of the pattern is to delegate the creation to a special factory object upon a client request. By using polymorphism, the factory object's class determines how the factory object reacts to a request for a certain object. E.g., when requested to create a window a MotifWidgetFactory object will return a Motif window, a PMWidgetFactory a PM-Window.

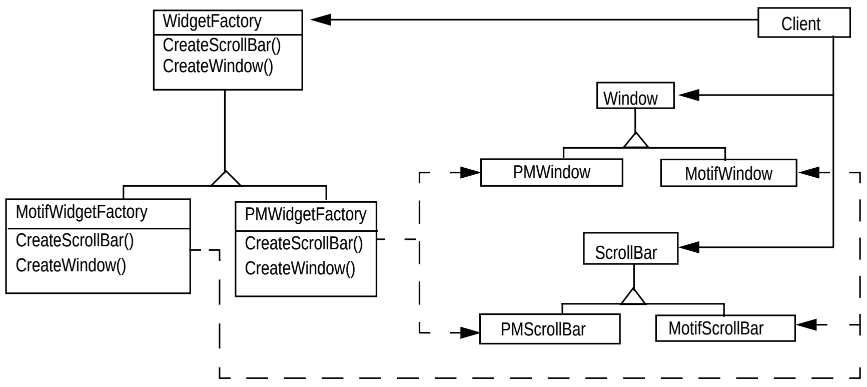

Figure 1. Illustration of the Abstract Factory pattern. We extend the OMT notation with dashed lines to indicate the relationship between the widget factories and the classes they relate. Note that this relationship will normally be hidden in the code.

3 Instantiating the Abstract Factory design pattern

In FACE, application development and adaptation is done at a higher level of abstraction than that of source code. It is done at the level of schema extension and editing. In figure 2 we see a diagram of a schema representing a typical instantiation of the Abstract Factory design pattern, corresponding to the example in figure 1.

We first make some remarks about the notation. The concept of a schema in FACE is related to the concept of a class-diagram in an object-oriented modeling technique such as OMT, since it intends to show the classes (entities that are or may be instantiated to run-time objects) and their relations. Thus, it is no surprise that the graphical notation uses elements similar to those used in class-diagrams of OMT. Standard relationships like subtyping (which is purely interface based sub-typing) and associations are used in FACE schemas, and occur in the figure. However as mentioned in the introduction, the schema for a specific pattern contains pattern specific relationships, in this case which

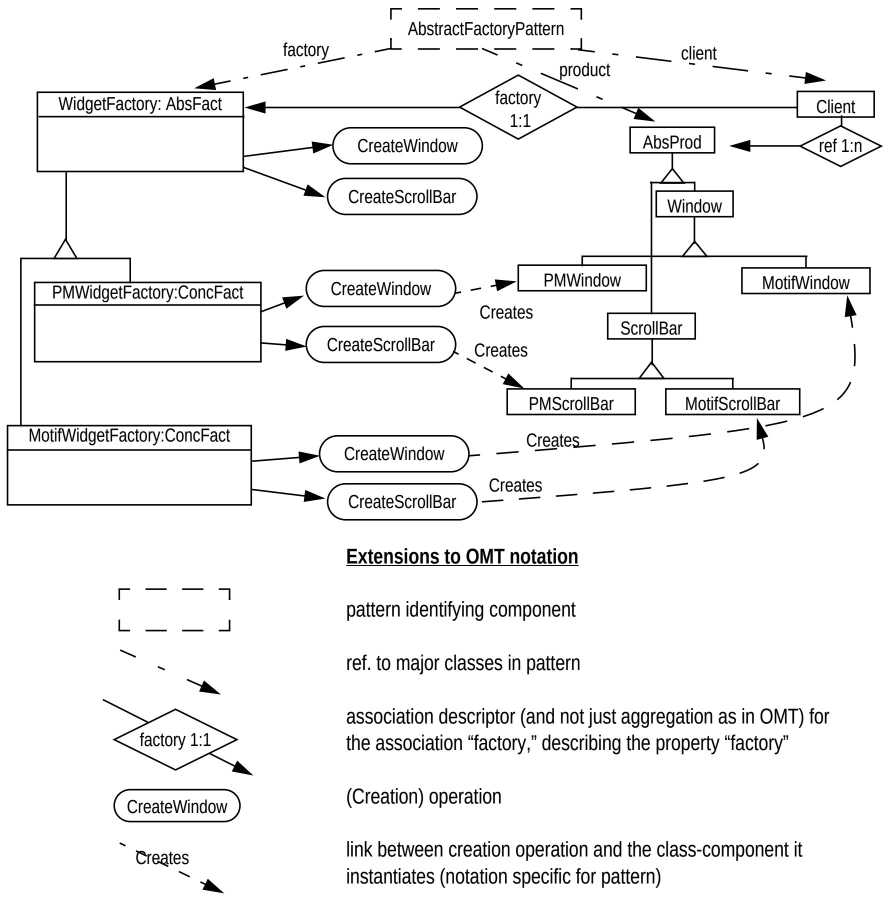
AbstractFactoryPattern
factory
client
product
WidgetFactory: AbsFact
factory 1:1
Client
AbsProd
ref 1:n
CreateWindow
CreateScrollBar
Window
PMWidgetFactory:ConcFact
CreateWindow
PMWindow
MotifWindow
Creates
CreateScrollBar
Creates
ScrollBar
PMScrollBar
MotifScrollBar
MotifWidgetFactory:ConcFact
CreateWindow
Creates
CreateScrollBar
Creates
Extensions to OMT notation
pattern identifying component
ref. to major classes in pattern
factory 1:1
association descriptor (and not just aggregation as in OMT) for the association "factory," describing the property "factory"
CreateWindow
(Creation) operation
Creates
link between creation operation and the class-component it instantiates (notation specific for pattern)

Figure 2. Diagram of a schema representing a typical instance of the AbstractFactory pattern, as corresponding to the example given in section 2.

operation instantiates which class. Such a modeling primitive falls outside the standard OMT modeling and thus has its own notation. Furthermore diamonds have a somewhat changed meaning: Diamonds do not automatically mean aggregation; they mean association. Some of the notational deviations are more conceptual and will be discussed below. We note explicitly that the diagram is meant to describe how a design pattern instantiation can be modeled in FACE. Such a notation might also be used by a corresponding tool, but this is currently not the case.

Before we go into detail regarding the elements and structure of such a schema, we need some special terminology. In general we shall speak of the components of the schema. We shall use the term "class-component"; one of the reasons for this terminology is that, for the application developers, classes in the schema are really black-box components that can only be specialized by means of parameters[1] and relationships. We now list some relevant aspects.

The first relevant aspect is the use of a separate component indicating that the Abstract Factory component has been used. It is also a container since it refers, using references named after the role that they play, to the most important abstract classes of the pattern. The class “AbsProd” in a sense plays a similar role: It is a container for the abstract product classes such as “Window” and “Scrollbar”; this will be used in the copying of the primal schema see section 4.

Secondly, the operations that form the heart of the pattern are promoted into explicit components of the schema, in this case the “create...” operations.

Thirdly, relationships are made explicit that have a context specific run-time meaning. In this case the operation “createWindow” of the class-component “MotifWidgetFactory” has an “instantiates” relationship with the class-component “MotifWindow,” indicating that this operation will create instances of that class.

Fourthly, class-components are typed as corresponding to the role they play in the pattern: For example, the class-components “MotifWidgetFactory” and “PMWidgetFactory” are both of the type “ConcFact” (Concrete Factory). This means that they are specialized in a specific way, namely in this case with the set of operations and per operation the link to the class-component that it creates and that this specialization has the corresponding run-time meaning (see section 8).

Finally, we made the association descriptor (as indicated using a diamond in the figure) as a separate independent schema component. One of the parameters of an association descriptor is whether the association is an ownership (and thus aggregation) or otherwise a reference. There are various both conceptual and FACE-implementation technical reasons for this which we will not further discuss.

We stress again that the schema makes only aspects explicit that are relevant: operations of the product classes that do not play a role in this pattern are not shown; internal structure is hidden, since it is of no relevance to the application developer. We thus require that product classes be implemented beforehand, either coming “ready made” with the framework or implemented in a separate coding phase of components.

In section 4, we show how we can define a meta-schema, what kind of elements (what kind of classes, what kind of relationships) a schema may have, and how these may be combined.

4 Framework specific definition of the Abstract Factory Pattern

In section 3 we saw that pattern specific class-components were used, and pattern specific relationships (e.g., “creates”) between these to form a schema. However, not all combinations of relationships and class-components lead to a correct “well-formed” schema. For example, the “creates” relationship must be made from a create operation to a concrete product class-component, and not to any other kind of class-component. In general, each pattern usage in a framework comes with a specific set of class-com-

1. Not shown in this example, parameters may also be procedures, thus allowing, where necessary, for greater flexibility.

ponents and relationships and with rules how these may be combined. This must be expressed in a so-called “meta-schema” to allow FACE to adapt to a specific framework with specific patterns. The meta-schema thus expresses the “modeling syntax” of the schema and will be discussed below. In section 9 we shall see how such a meta-schema may be used to support the application developer in creating a correct schema.

Just as an application developer using standard (“classical”) source code level application development starts from the set of abstract classes, the application developer in FACE does not start from scratch, but starts from a kernel schema, a schema that only contains the identifying component and the abstract class-components and their relationships, and adds concrete class-components and their relationships to this kernel. Therefore, a pattern definition not only encompasses the meta-schema (that defines how the kernel schema may be extended), but also a template for the kernel schema. This template is called the “primal schema.” These two together are represented together in figure 3.

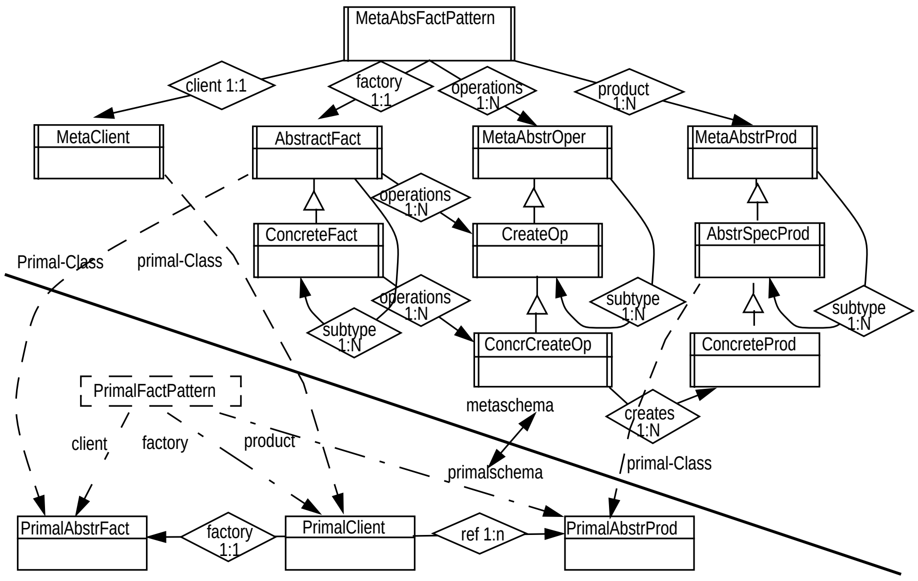

Figure 3. Abstract factory pattern definition as consisting of the meta-schema, and the primal-schema. The two are separated by a slash. Metaclass-components are indicated with double lines at the sides. These define classes of class-components, e.g., all Concrete Widget class components in figure 5 are instance of “ConcreteFact.” Association descriptors are again indicated by diamonds. Dashed lines are used to link a metaclass-component to its primal class-component.

The figure illustrates that the syntax rules themselves are also embodied as a schema, the “meta-schema.” This allows for the similar treatment of schemas and meta-schemas by tools (see section 9). The class-components in this meta-schema are called metaclass-components, the reason being that instances of a metaclass-component are class-components. The structural relationship between a metaclass-component and its instances is the same as between a class-component and its instances (see section 3). Thus, a meta-class component defines the existence and possible use of a certain type of special purpose class-components. In figure 3, for example, “ConcreteFact” defines the type of those class-components that correspond to factory objects. These class-components have “ConcFact” written behind their name in figure 2.

Most of the relationships between metaclass-components is via association descriptors as these were also described in section 3. These describe what kinds of structures are allowed in the schema. For instance the “creates” association descriptor between “ConcrCreateOp” and “ConcreteProd” describes that an operation such as “CreateWindow” has an explicit reference to the class-component that will be created by this operation (which must be a concrete product class-component). Through the “operations” association descriptor between “AbstractFact” and “CreateOp” is stated that Factories have these special kind of “creating” operations.

In figure 3 we also see the primal schema. This schema is copied, and used as the kernel for the rest of the schema development when the application developer instantiates “MetaAbsFactPattern.” In order to make the primal schema relatively general, “PrimalClient” is given an association with “PrimalAbstrProd” so that the Client object may refer to any kind of product.

5 Short Description of the State Design Pattern

The next 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 behavior, that is, dependent on its state the object reacts differently to the same requests for operation execution. The solution the pattern offers is illustrated in figure 4 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.

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.

6 Instantiating the State design pattern

In figure 5 a schema for a particular instantiation of the state design pattern is represented. This diagram is again close to a class-diagram, as in the example for the Abstract-Factory pattern (figure 2), it contains much more pattern specific information, so that it

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

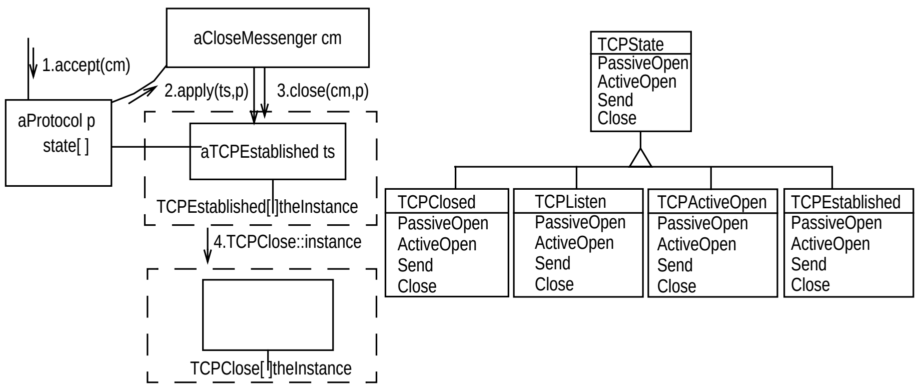

Figure 4. Solution to implementing the state design pattern as used by Hueni et.al.[5]. Left: An object p refers in its property "state" to one of the state objects. Each of these state objects is a "singleton" instance of the classes given at the right side. The figure shows a typical sequence of messages. 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 behavior 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.

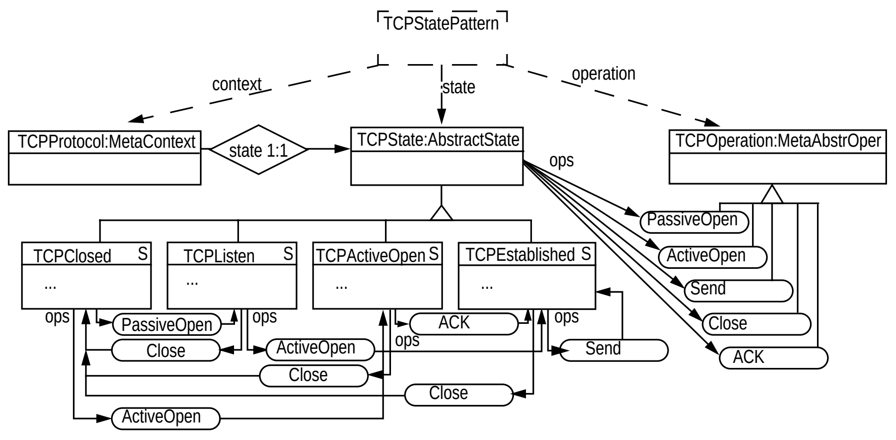

Figure 5. Diagram representing a schema that instantiates the state design pattern. In this case a realization of the TCP Protocol modeled in state pattern specific terms. The rectangles with a line in them are class-components. Class-components labelled with "S" represent concrete States. Ovals are again operations. The ones used to link two concrete state class-components are also called "transition descriptors." An arrows that links two concrete state class-components via a transition descriptor represents: when the operation named in the descriptor is applied to an instance of that class-component, the next state will be the one which the arrow points to.

describes in sufficient detail what the run-time system should do. Particularly, the enrichment encompasses:

- The use of framework (pattern) specific component types in the schema: for example all the concrete class-components that represent a state are of the type “ConcreteState.” This corresponds to a generic implementation. Another example are the transition descriptors like “Close,” “Send” that describe the links between the ConcreteState class-components.
- The use of framework (pattern) specific relationships between components of the schema: In particular, the links between ConcreteState class-components (as qualified by the transition descriptor) represent corresponding state transitions. The name of the transition descriptor corresponds to the operation that invokes this transition
- The transition descriptor may be parameterized with extra information that describes what else should happen in a state transition. Two such parameters are pre- and post- transition procedures, e.g. used for measuring response times. This is not shown.
- A pattern specific identifying component “TCPStatePattern” identifies the fact that the pattern is used; it is a container in the sense that it refers to all the major (abstract) class-components that play a role in the pattern.

As in figure 2, association descriptors are used here. For instance, the association descriptor between TCPProtocol and TCPState, describes that instances of TCPProtocol contain a reference to a state object. The association descriptor is thus a general component that can be used in any schema.

7 Framework-specific Definition of the State Pattern

Figure 6 illustrates how the state pattern may be defined as a meta-schema and linked to that a primal schema. Again the meta-schema embodies the rules for correctly creating schemas for this pattern and the primal schema defines the basic abstract class-structure that the application developer will use to extend. This primal schema is copied as a whole when the application developer instantiates the “MetaStatePattern.”

We note again:

- In the meta-schema metaclass-components specify what kind of class-components may occur, e.g., all the concrete State classes such as “TCPClosed,” “TCPListen” etc. are instances of ConcreteState, all transition descriptors are instance of “MetaTransDescr.” Association descriptors specify what kind of relationships these components may have, e.g., the association descriptor “operations” of ConcreteState describes that a ConcreteState class-components can have operations of the type “MetaTransDescr,” i.e., transition descriptors.
- Through the correspondence between meta-class components and primal classes, the correspondence between the meta-schema and the primal schema is made. In general, when the developer instantiates a metaclass-component, the resulting class-component is a copy of the primal-class of the metaclass-component. For

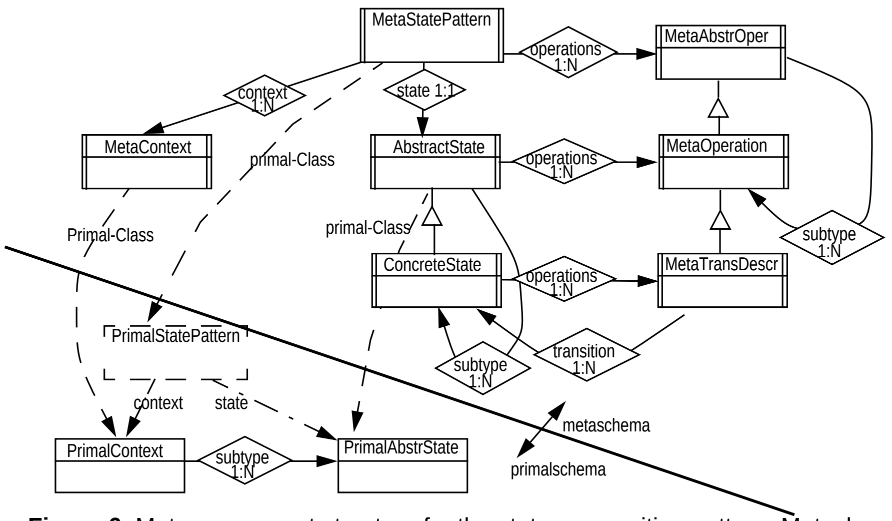

Figure 6. Meta-component structure for the state composition pattern. Metaclass-components are indicated with double lines at the sides. Association descriptors are again indicated by diamonds. See further the text and the explanation of figure 5 Note that since “MetaTransDescr” is a subtype of “MetaOperation,” transition descriptors, which are instances of MetaTransDescr are really operation descriptors.

MetaStatePattern, the whole structure consisting of PrimalStatePattern, Primal-Context and PrimalState is copied, since PrimalContext and PrimalState are structurally contained by PrimalStatePattern.

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

The idea of using schemas to describe applications has been described so far in an implementation independent way. One of the major questions is how the parameterization and linking of class-components and relationship components may lead to the corresponding run-time behaviour of the objects, and how we would allow such a correspondence to be easily (or as easy as possible) set up by the framework developer.

Basically two approaches are possible: a compiled approach in which “real” classes in standard object-oriented technology are generated from a schema or an interpreted approach in which the schema is represented explicitly (or “reified”) at run-time and the run-time software adapts its behavior to this information. Due to a greater simplicity and run-time configurability we have currently only applied the second approach. The first approach leads to more efficient execution, but this has not been a major concern so far.

In the second approach the schema is represented explicitly at run-time as objects that represent the class-components and relationships. This is quite natural, since the schema will normally be created interactively as an object structure by the application developer anyway. The class-components are thus run-time represented as objects but they function as classes, in the sense that they can be requested for instances. When requested for instances, they will return an instance object (either by copying a prototype that they carry or by calling a constructor). The instance objects will have a generic implementation, implemented in the underlying object-oriented language[1]. The implementation is generic in the sense that these instance objects will query the objects in the schema to adapt their behaviour to the parameters. Such a query will be called an "up-call." The class-component in its role of run-time accessible object thus implements behaviour to answer these queries.

We describe in further detail how the state class-components and their linking via transition descriptors determine the behaviour of the corresponding instances at run-time (figure 7).

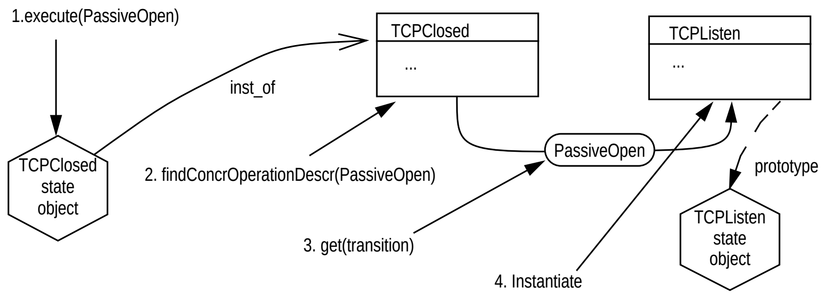

Figure 7. Object interaction for making a state object execute a transition diagram: The state object does an up-call to its class-component, in this case to "TCPClosed." "TCPClosed" finds the concrete operation descriptor for "PassiveOpen." The state object retrieves the information from this object, 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" and 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 7) 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. This descriptor is queried for its parameters. 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 ("TC-PListen" 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.

1. This underlying language can be anything: so far, implementations have been done in Self and C++.

9 Using a Meta-schema to Drive the Visual Composition Environment

So far we have shown through examples how a schema contains information for modeling a pattern in pattern specific terms, and how a meta-schema contains the modeling syntax for this. A FACE editing tool can actively support the application developer in modeling schemas correctly. Such a tool can be generic since it will be driven by the meta-schema.

It was suggested in section 1, and it is only natural, that the application developer should have *visual* support in creating and editing the schema. This strengthens the idea of explicitness: It gives immediate insight in (makes explicit) what components there are in the schema, what relationships there are between them, and thus also, what can be edited. Visual support should further help the developer in creating the right kinds of components, and creating the right kinds of relationships between them.

Currently, FACE uses the Kansas-Self [19] visual presentation to support visual modeling see figure 8. This means that (as mentioned in section 8) all components in a schema are represented as objects, and relationships as links between objects in Kansas. Kansas provides direct presentation and editing of these objects. The user of Kansas can also directly send messages to objects by entering the message in an evaluator dialogue box of the presented object. For example, the user can instantiate a meta-class component by directly sending the message “instantiate” to the object that represents it. To support meta-schema driven modeling only those meta-class components that need to be instantiated are presented, other information in the meta-schema is not shown. Relations between class-components are created by first using Kansas support to create a link between the objects that present those, and then sending a message to an object that represents the link to create the relationship; in doing so, a test will be made on basis of the meta-schema if the relationship is correct, otherwise it will be denied.

One could say that this is the “poor-man’s solution” of visual support; there are no “tailor made” presentations for schemas such as the ones shown in the figures in this paper, and the developer is not helped in performing a certain sequence of actions. We admit this but on the other hand we assert that it provides a proof of concept as it contains all the elements presented in this paper. Creating instances of a class or meta-class component, corresponds to copying the prototype it carries as described in section 8, which is again natural in Self. Using the schemas to determine run-time behaviour following the principle described in section 8 is no problem in the Kansas environment. Furthermore we assert that fancier presentations will not change the principle of supporting the developer in creating the correct components and relationships.

The principle of checking relationships that the developer attempts to create on the basis of the meta-schema is as follows: The system is implemented in such a way that for an attempted relationship (e.g., “creates” between the “CreateWindow” operation of “PMWidgetFactory” and the “PMWindow” product class in figure 2) the corresponding association descriptor can be found (in this case the “creates” association descriptor as presented in figure 3). This association descriptor is queried by the method that checks the attempt. A check is made whether the target of the link is an instance of the

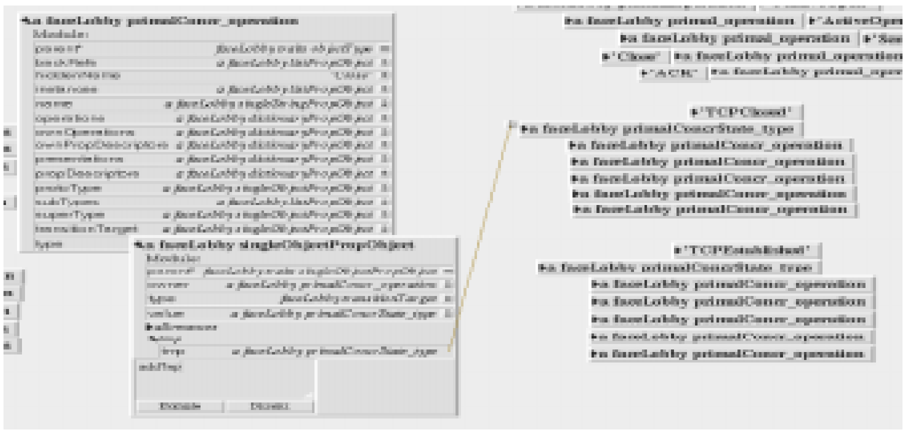
a faceLobby primalConcr_operation
a faceLobby primal_operation
'Close'
'ACK'
'TCPClosed'
a faceLobby primalConcrState_type
'TCPEstablished'
a faceLobby singleObjectPropObject
tmp
addTmp

Figure 8. Snapshot of FACE Class-composition as made available through Kansas: Each class-component is represented by an object. Each property, e.g., “transitionTarget” (corresponding to the “transition” property described in figure 6) is itself represented by an object. Linking the tmp attribute of that object to another object corresponds to “attempting” to make the link between the owner of the property object and the other object, in this case the link is attempted between a concrete operation component for the transition ‘Close’ for the “transitionTarget” property to a concrete state class-component, namely the one that represents TCPClosed. By requesting evaluation of “addTmp” (push the “Evaluate” button) the composition will be made if it is a correct link, as is the case here.

meta-class component that the association descriptor points to or of one of its subtypes. If this check returns true, the attempt succeeds. In this example, “PMWindow” is an instance of “ConcreteProd,” so the attempt succeeds.

10 Related work

The idea of actively supporting design patterns is finding its way into CASE tools. A first commercial product supporting patterns is Objectif [12]. A more experimental approach is described by Pagel & Winter [15]. These kinds of tools focus on helping the framework designer: patterns are mainly used to support the design of the framework architecture. Code can be generated, but this is not the main goal of these tools.

Closer to our work is work is work described by Soukop [19] and Sommerlad, et.al. [17], which supports application developers. Basically these tools provide quite extensive code generation. In the case of Soukop, it is quite clear that the developer has to further adapt the classes after code has been generated and that thus the link between the higher abstraction (which are basically only macros) and the code will further be lost (this was called the design-implementation gap problem in the introduction). In the work of Sommerlad et.al, it is not clear whether the higher abstraction level allows for adding code, and thus precludes having to change generated code. We assume that they have the same design -implementation problem. They certainly have the problem that creating framework-specific support is a heavy burden on the framework developer. In FACE, no tool extension or adaptation is needed: only the adaptation or extension of meta-schemas.

Vista [11] is a visual tool that supports composition, and that can be adapted to the composition rules. In contrast to FACE, Vista does not support a precise distinction between class components and instances.

Composition at the level of classes (although not visually supported) is encountered in particular in "generic" constructs, where classes can be parameterized with other classes, such as in the template classes in C++. The Standard Template Library STL [13] 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 [10] represents another form of class-level composition: Links between components basically describe communication possibilities between the instances. All these approaches offer a fixed meaning to links between classes. In contrast, in FACE arbitrary kinds of relationships may be introduced. This is a result of the meta-level that is provided.

In this respect, our work has roots in the area of open programming languages such as CLOS[6], and in Open Implementations[16]. Open programming languages such as CLOS reify their software components (classes), but the reification is not "black-box" the reification is quite 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 [7]. Their model also allows one to introduce new kinds of relationships. They focus on adaptive data models for databases systems, less on frameworks and application development. We feel that our model is simpler and therefore easier to understand and use.

Steyaert et. al. [20] have described how the use of a meta-level interface could be applied to provide powerful configuration capabilities for user interface builders. Also, using reflection, new kinds of components could be described using the composition environment itself. However, the way in which they open up the framework and corresponding visual composition seems to be restricted: They do not describe mechanisms for introducing new component cooperation forms in the framework.

The work of Lieberherr et. al.[8][9] 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 also cover traversal adaptiveness (but we still have to prove that), although it is, of course, not as well "tuned" for that.

11 Conclusion and Further work

In most software development environments, there is a gap between the design level, where new CASE tools may let the designer represent the system in terms of design patterns, and the implementation level, where the developer works at the level of source code. In working at the implementation level the connection with the higher-level concepts is either implicit or only available in terms of documentation, so that adapting and extending a system in agreement with the chosen design patterns is not actively supported. We call this problem the design-implementation gap. We have shown in this paper how to represent software as a "model" at the high-level abstraction of design patterns, while still capturing enough information for running the application. We have thus presented a “modeling = programming” approach. Basically, the model (called a “schema”) makes explicit all the information that the application developer may want to see and/or change. Such information is normally implicit in source code, such as the roles of classes and relation between the classes. All other implementation information remains implicit. Furthermore we have shown that a visual composition tool, for which FACE currently implements a simple solution, can help the developer to create and adapt these models correctly according to the pattern specific syntax rules.

By illustrating such a modeling = programming approach for design patterns, we imply that the same approach will carry over to full frameworks consisting of several patterns and other components and relationships. Still, how this would work is outside the scope of this paper. We stress furthermore that we have focussed on raising the abstraction level to the level of design patterns, not on making instantiations of design patterns into modeling “molecules” that can be reused blindly in any framework. The reason is that each framework applies design patterns in a specific way.

FACE is a framework adaptive “environment.” It can be adapted to different frameworks and in more restrained sense to the patterns in the framework by means of a meta-schema and generic software that implements the semantics of a schema. We have shown how this may be done. It is our hypothesis that due to this technology the gain of simpler application development is not lost by the effort setting up and adapting such a “modeling = programming” framework and that this approach will thus be an attractive alternative to “standard” object-oriented framework technology. This still needs to be proven.

Future work will be in two directions:

- Using FACE for re-engineering purposes [19]: discovering patterns, anti-patterns, the relationships between classes etc. and representing this in FACE schemas; afterwards, improve system design by applying restructuring transformations based on design patterns.
- Applying FACE to real world frameworks. Interesting questions will be, how complex meta-schemas will become. We want to investigate whether we can uncouple patterns (similar as described by Soukop [19]) so that the reuse and merging of patterns will be easy.

Acknowledgements

We thank Dirk Riehle for his first reviews; Tamar Richner, Robb Nebbe, Oscar Nierstrasz and many others for their helpful comments and corrections. This work has been carried out at the Software Composition Group, University of Berne, and funded by the Swiss National Science Foundation under Project no. 2000-46947.96.

References

[1] Don Batory and Sean O’Malley, “The Design and Implementation of Hierarchical Software Systems With Reusable Components,” *ACM Transactions on Software Engineering 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] Serge Demeyer, Stéphane Ducasse, Robb Nebbe, Oscar Nierstrasz and Tamar Richner, “Using Restructuring Transformations to Reengineer Object-Oriented Systems,” Submitted to WCRE’97. Available from the SCG-website (http://iamwww.unibe.ch/~famoos/).

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

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

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

[9] 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.

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

[11] Vicki de Mey, “Visual Composition of Software Applications,” *in [14],* pp. 275-303.

[12] Microtool homepage: http://www.microtool.de/

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

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

[15] Bernd-Uwe Pagel, Mario Winter, “Towards Pattern-Based Tools,” *EuroPLoP Preliminary Confer*ence Proceedings, July 1996

[16] 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.

[17] Albert Schappert, Peter Sommerlad and Wolfgang Pree, “Automated Support for Software Development with Frameworks,” *Proceedings SSR’95 ACM SIGSOFT Symposium on Software Reusability*, 1995.

[18] Randall B. Smith and David Ungar, “Programming as an Experience: The Inspiration for Self,” *Proceedings ECOOP’95,* W. Olthoff (Ed.), LNCS 952, Springer-Verlag, Aarhus, Denmark, August 1995, pp. 303-330.

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

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