Class Composition for Specifying Framework Design[1]

Serge Demeyer, Matthias Rieger, Theo Dirk Meijler, Edzard Gelsema

Abstract: Object-oriented frameworks are a particularly appealing approach towards software reuse. An object-oriented framework represents a design for a family of applications, where variations in the application domain are tackled by filling in the so-called hot spots. However, experience has shown that the current object-oriented mechanisms (class inheritance and object composition) are not able to elegantly support the "fill in the hot spot" idea. This paper introduces *class composition* as a more productive approach towards hot spots, offering all of the advantages of both class inheritance and object composition but involving extra work for the framework designer.

Introduction

Current software technology has to cope with many challenges. Software is more and more becoming a crucial part of vital business processes, thus must be correct and robust. Applications get more complex (i.e., GUI, inter- and intranets), but the development effort has to be kept minimal to maintain low prices. Finally, software is used in ever changing circumstances, thus has to be easily adaptable.

These challenges are not new and have motivated the search for reusable software. Indeed, reuse has the potential to improve software correctness (reused software is better tested) and robustness (reuse in different contexts ameliorates stability), while shortening the development time (software does not need to be developed again). Indirectly, reuse may also help in coping with complexity (complex entities can be composed from reusable assets) and ease the adaptability (designing adaptable items pays off when they are reused).

With respect to software reuse, both research and industry pay a lot of attention to object-oriented application frameworks [JohnsonFoote'88], [FayadSchmidt'97]. A framework is then defined as an artefact that represents a design to be reused in a family of applications. Framework designers specify variations within the framework design by means of hot spots; application developers refine the framework design to the needs of their application by filling in those hot spots.

Object-oriented languages provide two mechanisms for defining and filling in hot spots: *class inheritance* and *object composition* [GammaEtAl'95]. The former mechanism applies subclassing and overriding to refine the framework design and is often referred to as being "white box" since subclasses have access to the internals of their parent classes. The latter mechanism relies on late binding polymorphism to vary the actual method invoked on another object. Object composition is often called "black box", since objects can only use the external

[1] Theory and Practice of Object Systems (TAPOS), vol. 5, no. 2, April 1999, pp. 73-81.

interfaces declared by the other objects. Experience has shown that both mechanisms have their benefits and drawbacks; we listed them in the following table (benefits are marked with a '+', while drawbacks are marked with a '-').

Class inheritance (white box hot spot)		**Object composition (black box hot spot)**	
+	Discernible. Class inheritance hot spots are defined via abstract method invocations, which are easy to recognise in class diagrams and in source code.	-	Concealed. Object composition hot spots are defined via polymorphic method invocations, difficult to distinguish in source code and class diagrams without proper documentation.
+	Elementary. Creating a new subclass and overriding a method is close to an atomic operation within any object-oriented modelling language.	-	Complicated. Plugging in a new object involves a series of steps. Without proper tool support, a framework becomes vulnerable to configuration mistakes.
-	Jeopardised. Inheritance exposes a subclass to implementation details of its parent classes, which easily leads to undesirable implementation dependencies.	+	Encapsulated. Object composition requires objects to respect each others interfaces which makes it easier to hide the object's internal implementation details.
-	Rigid. Creating a new subclass involves some form of recompilation. Run-time system extensions require dynamic linking.	+	Run-time Extensible. Plugging in a new object can be done while the system is running without any recompiliation or re-linking..

The benefits of one mechanism seem to complement the drawbacks of the other, thus framework designers are free to choose how to define a hot spot. However, run-time extensibility is becoming a stringent requirement in today's software systems (see for instance [LaddagaVeitch'97]) and this is better supported through object composition. Nevertheless, this extra capability is quite expensive since filling in an object composition hot spot is a complicated procedure which puts the burden on the application developer. This is counterproductive: as one framework is supposed to be reused in many applications, the work load should be on the framework designer and not on the application developer.

This paper introduces *class composition* as a mechanism that sits in between class inheritance and object composition. Class composition involves a meta-modelling step to produce a parameterisable class model. This way a framework designer can offer an application developer the combined benefits of class inheritance and object composition (i.e., discernible, elementary, encapsulated and run-time extensible). The combined benefits involve extra work for the framework designer — a productive way to achieve those benefits if a framework is reused in many applications.

We start the paper with an example which illustrates the benefits and drawbacks of class inheritance and object composition. Then, class composition is applied on the same example, showing that class composition indeed offers all of the claimed benefits in a productive way. Afterwards, the paper provides an overview of other occasions where we have employed the notion of class composition and tells about our plans for the future. Before coming to a conclusion, the paper reports on related work within the object-oriented community.

Hot Spots with Class Inheritance

To contrast class composition with former techniques for framework design, we use an experimental framework for on-the-fly generation of HTML pages. The framework is intended to support the maintenance of a world-wide web site for a corporate intranet. The purpose of such a web site is to provide up to date information, thus the pages must be generated based on information maintained in a number of existing databases. A typical element of such a web site is for instance the corporate telephone directory, which might be maintained in some legacy database system able to dump a tab-separated text file every week. Another part of the web site is a page for asking which employees work for a given project; that volatile information is presumably available from a relational database maintained by the accounting department. Databases are therefore an important variation in our application domain, and thus our framework should cope with this variation.

Frameworks tackle variation in the application domain via a hot spot, thus our framework should at least include one for "Database". Figure 1 shows a class diagram for a class inheritance hot spot, realised via a template method [GammaEtAl'95]. The hot spot is specified in the template method `generateHTML` defined on the class `Database`, which invokes first the abstract method `fetchTable`, and afterwards enumerates the records in the returned table to render the corresponding HTML. To fill in the hot spot, application developers must subclass `Database`, providing an implementation for the hook method `fetchTable`. In our example, we provide a first subclass `PhoneDatabase` which implements the `fetchTable` method by filtering information out of a text file; the filename and some wildcard operator is passed via the `tableSpec` argument. The second subclass `ProjectDatabase`, implements the `fetchTable` method by opening a database connection, sending an SQL query based on the `tableSpec` argument, translating the result of the query into an instance of `Table` and closing the database connection.

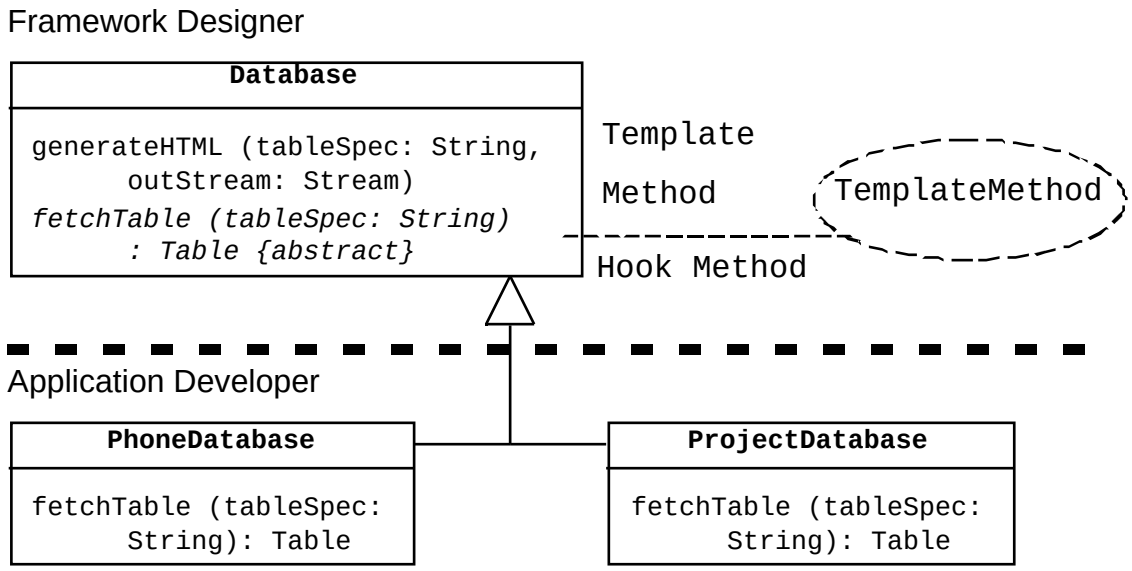

Figure 1: Framework class diagram with the "Database" hot spot defined via a template method. The hot spot is filled in by the subclasses `PhoneDatabase` and `ProjectDatabase`, which override the abstract method `fetchTable`.

To incorporate the various databases into the web site, the application developer creates a CGI-script that invokes the `generateHTML` method on an instance of `PhoneDatabase` and `ProjectDatabase` with the appropriate parameters. This way, the application developer reuses the application logic of the framework, defined in the template method `generateHTML`.

So far, the example has illustrated the basic roles and steps applied in all frameworks:

Framework Designer

1. *Identify a variation* in the application domain. In the example, the database maintaining the information.
2. *Define a corresponding hot spot* via a template method. In the example method `generateHTML` invoking the abstract method `fetchTable`.

Application Developer

1. *Fill in the hot spot* by providing a hook method. In the example by providing a subclass overriding the method `fetchTable`.
2. *Reuse the framework logic* defined in a template method. In the example, the sequence of method invocations as is defined in the method `generateHTML`.

With the above example, the approach works pretty well, because the class diagram makes the hot spot immediately visible (the abstract declaration) and because application developers can readily fill in the hot spot (create a subclass + override the abstract method). However, the fact that subclasses have access to the implementation details of their parent classes introduces the potential problem that a subclass may inadvertently break the design of the framework (e.g., override `generateHTML` without invoking `fetchTable`). Also, because of the compile-time nature of inheritance relationships, we cannot extend the system at run-time (e.g., add a new database subclass). Especially the latter may be quite cumbersome, and the next section shows how object composition hot spots circumvent that problem.

Hot Spots with Object Composition

In a geographically distributed corporation, it is unlikely that all desktop machines will run the same web browser. For optimal page layouts, one must tune the HTML generation towards the requesting browser. For instance, when the requesting browser accepts HTML 3.0, the web page should use the `<table>` tags to render tabular information. However, for all browsers understanding a lower version of HTML, web pages must mimic tables via a `<pre>` tag and align the columns via spaces (see Figure 2). Thus, rendering HTML is another variation in the application domain.

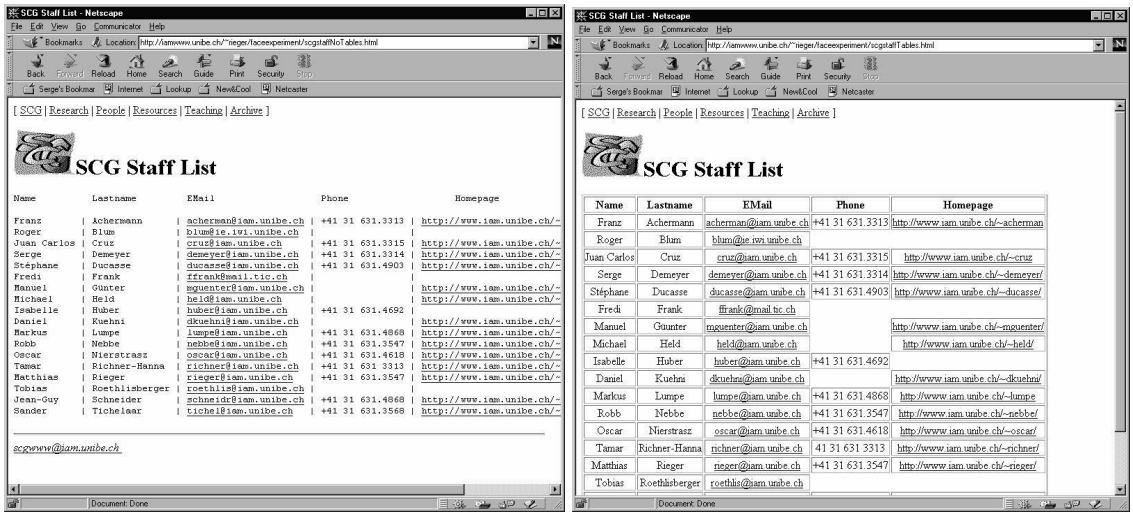

Figure 2: Screen dump of a web-browser displaying a table in HTML 2.0 (with space padding) and HTML 3.0 (with table tags)

Applying the framework principle for variations in the application domain, the framework designer must define a hot spot by means of a template method. An instinctive attempt would be to extend the `generateHTML` method, having it invoke another abstract method named `renderHTML`, also defined on the class `Database`. By subclassing and overriding this `renderHTML` method, an application developer would then provide the appropriate algorithm for generating HTML. However, there are two reasons why this solution is not satisfactory. First, the `Database` class violates the "separation of concerns" principle as it implements two variations of our application domain. One variation for database extraction and another for generating HTML. To mix both implementations, an application developer is obliged to apply multiple inheritance; quite disputable and in some object-oriented languages not always possible. Second, for each new `renderHTML` operation, one must create a new subclass, which involves recompiling/re-linking the system and thus implies a temporary shut down of the web server or dynamic linking technology. Since web standards and browsers evolve quite rapidly, there will be a frequent need for new HTML renderings and thus system extensions should be much easier. Current applications —especially on the web— often demand for run-time extensions [LaddagaVeitch'97] in frameworks achieved by means of object composition hot spots.

To define an object composition hot spot a framework designer does basically the same as in the case of class inheritance, i.e. define a template method. The only difference is that the

template method now invokes a hook method defined on a class which is different from the one defining the template method. This way, the varying behaviour is factored out into a separate class allowing the hot spot to benefit from late binding polymorphism. Figure 3 applies this to our HTML generation example; it shows how the refactored `generateHTML` template method now also invokes the hook method `renderHTML` defined on a new class called `HTMLRenderer`.

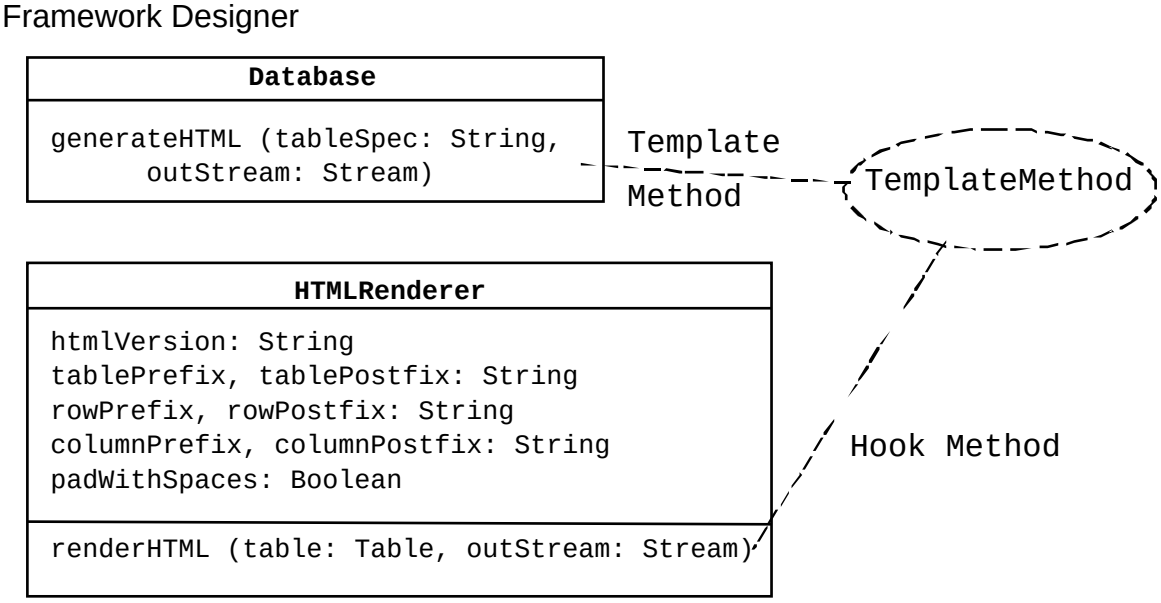

Figure 3: Framework class diagram defining the "renderHTML" hot spot. The hot spot must be filled in by creating instances of the `HTMLRenderer` class.

Filling in an object composition hot spot is entirely different from filling in a class inheritance hot spot however. Rather than creating a subclass and overriding a method, an application developer (1) creates an object; (2) sets the appropriate state; and finally (3) links the object with the appropriate data structures. Figure 4 fills in the hot spot for our example, by (1) creating two instances of the `HTMLRenderer` class (`htmlRenderer20` and `htmlRenderer30`); (2) storing the parameters of the generation process in the attributes of these objects (i.e., set the `tablePrefix` to `'<PRE>'` or `'<TABLE>'` and `tablePostfix` to `'</PRE>'` or `'</TABLE>'`); (3) installing the object into the global table `ApplicableRendering` (a table that maps a HTML version number on an instance of `HTMLRenderer`). When the hot spot is filled in correctly —especially linking the objects with the appropriate data structures can be quite error prone when not supported by tools— the application developer reuses the framework logic as specified in the template method. In our example, this includes the consultation of the global `ApplicableRenderings` table to look up the best `htmlRenderer` for the given version of HTML.

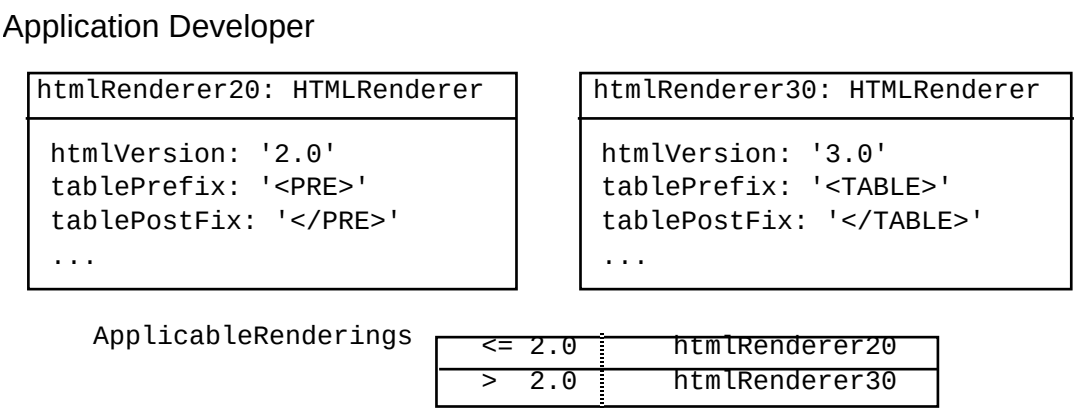

Figure 4: Framework object diagram filling in the "renderHTML" hot spot. The instances `htmlRenderer20` and `htmlRenderer30` are stored in the global table `ApplicableRenderings`, which maps a HTML version number on the appropriate object.

Note that filling in an object composition hot spot does not necessarily imply compilation —creating objects, setting state and linking with other objects can for instance be performed via a script or via a composition environment— and thus allow run-time extensions of the system. Also, since a hook method and a template method are defined on distinct classes, both methods can rely on each other's public interfaces only, resulting in better encapsulation. However, object composition hot spots manifest some notable drawbacks compared to class inheritance hot spots as well. First of all, the hot spot itself is not as visible in the class diagram (or in source code) because the hook method is not different from any other method (for instance, it's not declared abstractly). This explains why object composition hot spots must be documented more thoroughly, i.e. by means of cookbooks and design patterns [Johnson'92]. Second, filling in the hot spot is a more elaborate procedure. In particular, linking the object in the appropriate data structures can easily lead to erroneous software if not backed up by the appropriate composition tools, i.e. by means of composition environments [deMey'95] and typing. Since an object composition hot spot is less discernible and since filling in a hot spot is more complicated, object composition is somehow a counterproductive extension mechanism: it puts the burden on the application developer and not on the framework designer.

Thus object composition hot spots can eliminate the drawbacks of class inheritance hot spots (jeopardised, rigid), but consequently also prohibit the immediate benefits (discernible and elementary) and involve extra work for the application developer. The next section introduces class composition as a technique that offers the advantages of both class inheritance and object composition in a productive way.

Hot spots with Class Composition

Reconsidering the diagram in Figure 3, we see that the class model does not express the core design of the framework. Especially the framework specific relationship between the class `Database` and the class `HTMLRenderer` is entirely lost — this relationship is represented by means of the `ApplicableRenderings` table, a global variable not represented in the

class diagram. Yet, this special relationship is the heart of the `renderHTML` hot spot, and the class diagram should be adapted to include that framework specific relation.

However, incorporating dynamic object relationships into static class diagrams is not evident; special techniques are necessary to perform such a task. In the case of class composition, we apply meta-level modelling [KiczalesEtAl'91], [Rao'91]. That is, we extend the primitive modelling constructs provided by our underlying modelling language to specify parameterisable class models defining the specific relationships needed by the framework. The stepwise procedure below clarifies how this works in practice.

To define class composition hot spots for the HTML generation framework, the framework designer first specifies two special meta-classes (`ConceptualType` and `ConceptualOperation`) parameterised by one meta-relationship (`ApplicableOperation`). `ConceptualType` and `ConceptualOperation` are meta-classes serving as placeholders for a list of properties (i.e., the state) and a list of operations (i.e., the behaviour). However, `ConceptualType` defines all that belongs to the core of the framework and can never be changed by the application developer, while `ConceptualOperation` defines all that is application specific and must be supplied by the application developer. `ApplicableOperation` is then the relationship between those two meta-classes, telling which `ConceptualOperation` can be applied on which `ConceptualClass`.[2]

In a second step, the framework designer instantiates the above meta-model to derive a class model where the class composition hot spots are defined by means of framework specific class relationships. `Database`, `HTMLRenderer` and `FetchTable` are all classes because they are instances of the meta-classes `ConceptualType` or `ConceptualOperation`. However, `Database` defines the framework specific operation `generateHTML` to be invoked by the application developer. Its implementation is provided by the framework designer and can never be changed by the application developer. On the other hand, `FetchTable` and `HTMLRenderer` define the application specific database extraction and HTML generation operations including the necessary attributes. An application developer must provide the values of the attributes and the implementation of the operations. Figure 5 shows how this looks like in UML, using stereo-types as the meta-level extension mechanism.[3]

[2] Note that all meta-level extensions are expressed by means of the primitive constructs provided by the underlying host language. In the experiment described here we use C++ and UML, but we have experiences with Self as a host language as well.

[3] Stereotypes are depicted by means of guillemets (<<>>) and represent a built-in extension mechanism of UML [BoochEtAl'96].

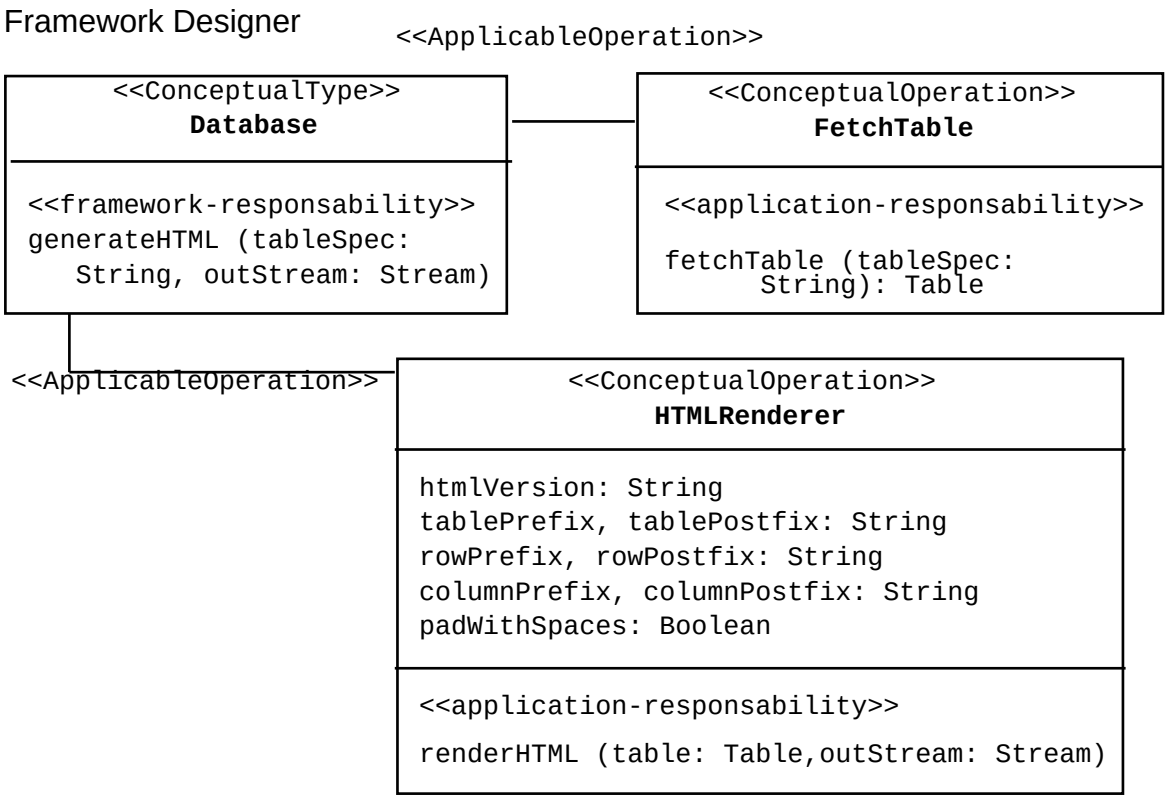

Figure 5: Framework class diagram with the "renderHTML" and "fetchTable" class composition hot spot. The hot spots are qualified with the parameterised relation `ApplicableOperation`.

To fill in class composition hot spots, an application developer creates instances of the appropriate classes (i.e. instances of the meta-classes) and links them together via the appropriate relationships (i.e., instances of the meta-relations). In our example, an application developer will create one instance of `Database`, two instances of `FetchTable` (once for the `PhoneDatabase` and once for the `ProjectDatabase`) and two instances of `HTMLRenderer` (one for HTML version 2.0 and one for HTML version 3.0). Afterwards, the `ApplicableOperations` relationship has to be instantiated, i.e. an application developer must associate the created instances in the appropriate way.

With a class composition hot spot, we achieve the combined benefits of class inheritance and object composition hot spots. Firstly, a class composition hot spot is almost as discernible as a class inheritance hot spot. In a class diagram it is immediately visible via the stereotyped classes and associations; in source code it will show up as an instance of a meta-class. Secondly, the meta-level support turns the filling in of a class composition hot spot into an elementary step for the application developer. Filling in a class composition hot spot corresponds to instantiating the classes and defining the proper associations. Thirdly, a class composition hot spot preserves encapsulation, because the framework designer can shield the core design of the framework so that it is never modifiable by the application developer. Finally, class composition is as run-time extensible as object composition, because instantiating meta-classes is no different than instantiating normal classes — it can be done from a scripting language or composition environment.

However, class composition changes the basic roles and steps in frameworks:

Framework Designer

1. *Identify a variation* in the application domain. This is the same as before.
2. *Define meta-level extensions*. In the example, the meta-classes `ConceptualType`, `ConceptualOperation` plus the meta-relationship `ApplicableOperation`.
3. *Define a corresponding hot spot* via instantiating the meta-level. In the example the classes `Database`, `HTMLRenderer` and `FetchTable` with the `ApplicableOperation` relationship between them.

Application Developer

1. *Fill in the hot spot* by creating instances of classes and creating the appropriate associations.
2. *Reuse the framework logic* defined in a framework specific method. In our example, the sequence of method invocations like it is defined in the method `generateHTML`.

Compared to framework design with normal hot spots, this implies considerable extra work for the framework designer. If one framework design is reused in enough applications, this extra cost surely outweighs the benefits. Moreover, it is possible to reuse the same meta-level extensions across different frameworks. For instance, we have reused the `ConceptualType` and `ConceptualOperation` meta-classes in a framework for generating web-pages (the experiment described in this paper) and for integrating software [Meijler'93]. Also, we have done work on defining a library of meta-level extensions for expressing design patterns to be reused across frameworks [MeijlerEtAl'97b].

Past, Present and Future

The initial ideas on class composition were developed during the Ph.D. work on the Yanus system [Meijler'93]. Yanus was a design for an integration system running in a medical environment where patient data coming from various sources were to be analysed by diverse statistical packages. Yanus modelled this in a generic way via an open set of databases supplying data to be manipulated by an open set of applications. Traditional object-oriented techniques were not sufficient to adequately model the required genericity, which explains the search for a reflective data model.

Achieving genericity via meta-modelling is not easy: the constant transition between the meta-level, framework level and application level requires a suitable composition environment. This has inspired the work on FACE (Framework Adaptive Composition Environment) [MeijlerEtAl'97a], [MeijlerEtAl'97b] which implemented the Yanus model in Self and provided a complete visual composition environment nicely embedded within Kansas-Self [SmithUngar'95]. To validate class composition as a way to produce parameterisable class models, FACE was targeted towards a library of design patterns [GammaEtAl'95]. Indeed, since each framework incorporates its own variations of the standard design pattern catalogue, one needs parameterisable class models to reuse the same model across different frameworks. FACE showed that class composition was able to offer the required adaptability. Other work on parameterisations, frameworks and hot spots has been reported in [DemeyerEtAl'97b], where we presented how objects representing framework contracts can provide tailorable frameworks.

To show how class composition can be achieved with "normal" object-oriented means, we implemented the FACE model in C++ [Rieger'97]. It is this implementation that was used to set up the experimental web-site described in this paper. In order to implement FACE with C++, we created our own object system by means of generic STL containers. On top of that we implemented the FACE type system including inheritance- and instantiation mechanisms. To hook a normal C++ class into the FACE system it suffices to subclass the abstract superclass that implements the necessary FACE type behaviour and link the C++ class to a special purpose FACE type. A FACE object is then instantiated by the FACE type which delegates part of the instantiation to the corresponding C++ class. In this way the FACE object system is mapped onto the C++ object system.

Currently, the notion of class composition is used within a generic work flow system within Baan Labs. Preliminary results lead again to the conclusion that class composition is a an adequate mechanism to define parameterisable class models.

Finally, we plan to apply our FACE experience with libraries of design patterns to the problem of object-oriented re-engineering [DemeyerEtAl'97a]. Using a library of anti-patterns/patterns, we would detect well-known object-oriented anomalies and transform them into the more appropriate programming constructs.

Related Work

Several authors have reported parameterisations of class models for better support of reusability and frameworks, sharing ideas and intuitions with our work.

First of all, the object-oriented community has extensively studied reflective programming languages to support extensibility. CLOS [KiczalesEtAl'91] is one of the more eye-catching examples of that line of work. Reflective programming languages have proven their usefulness, but their customisation capabilities are targeted towards programming language semantics, which makes them too technical for expressing reusable framework designs. This explains why researchers have tried to find more generic and less controlled way for tailoring the expressiveness of the programming language. Adaptive programming [Lieberherr'94], composition filters [AksitEtAl'92], aspect-oriented programming [KiczalesEtAl'97] and executable connectors [DucasseRichner'97] belong to that category, as does our own work on class composition.

In the area of object-oriented databases, research has been conducted to experiment with parameterisable class models. Indeed, the schema of a database is quite similar to our notion of class composition, i.e. a declarative description of the objects and relationships that may occur at instance level. VODAK [KlasEtAl'90] is an example of an object-oriented database which uses meta-classes to specify parameterisable database schemas. Filling in those parameters produces an application specific schema which is in turn instantiated to populate the database. Composition environments have also tackled the problem of parameterisable class models. Vista [deMey'95] is a composition environment where the rules for composition as well as what can be composed is variable. ApplFLab [SteyaertEtAl'96] showed that reflection can be used to parameterise visual user interface builders.

Conclusion

We have introduced *class composition* as a new mechanism for defining and filling in hot spots. Class composition sits in between the two mechanisms currently known within object-oriented frameworks: class inheritance and object composition. Class composition offers an application developer all of the advantages of class inheritance and object composition at the cost of extra work for the framework designer. If a framework design is reused in quite a large number of applications, class composition is a cost effective mechanism.

We have applied class composition in a number of framework related experiments, using implementation languages like C++ and Self. From those experiments, we conclude that class composition is possible with today's object-oriented technology. However, we are working on tool support to enhance class composition, especially in the area of visual composition environments and design patterns. With the appropriate tool support, we are optimistic that class composition can bring a substantial improvement for today's framework industry.

Acknowledgements

This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975.

References

[AksitEtAl'92] Aksit, M., Bergmans, L., and Vural, S, An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In Proceedings ECOOP'92, O. Lehrmann Madsen (Ed.), LNCS 615, Springer-Verlag, Utrecht, The Netherlands, June/July 1992, pp. 372-39

[BoochEtAl'96] Booch, G., Jacobson, I. and Rumbaugh, J, The Unified Modelling Language for Object-Oriented Development. See http://www.rational.com/.

[deMey'95] de Mey, V., Visual Composition of Software Applications. In Nierstrasz, O., Tsichritzis, D. (Ed.), Object-Oriented Software Composition, Prentice Hall, 1995.

[DemeyerEtAl'97a] Demeyer, S., Meijler, T. D. and Rieger, M. Towards Design Pattern Transformations. FAMOOS Workshop on Object-Oriented Software Evolution and Re-Engineering, organised with ECOOP'97 Conference. To appear in ECOOP'97 workshop reader.

[DemeyerEtAl'97b] Demeyer, S., Meijler, T. D., Nierstrasz, O. and Steyaert, P., Design Guidelines for Tailorable Frameworks. Communications of the ACM 40, 10 (October 1997), pp. 60-64

[DucasseRichner'97] Ducasse, S. and Richner, T., Executable Connectors: Towards Reusable Design Elements. In Proceedings of ESEC/FSE'97, LNCS 1301, 1997, pp. 483-500.

[FayadSchmidt'97] Fayad, M. and Schmidt, D. C., Object-Oriented Application Frameworks. Introduction to a special issue of the Communications of the ACM 40, 10 (October 1997), pp. 32-38.

[GammaEtAl'95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns. Addison-Wesley, 1995.

[Johnson'92] Johnson, R, Documenting Frameworks Using Patterns. In Proceedings OOPSLA'92, ACM Press, October 1992.

[JohnsonFoote'88] Johnson, R. E. and Foote, B., Designing Reusable Classes. Journal of Object-Oriented Programming 1, 2 (February 1988), 22-35.

[KiczalesEtAl'97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming. Proceedings ECOOP'97, Aksit, M. and Matsuoka, S. (Ed.), LNCS 1241, Springer-Verlag, July 1997, pp. 220-242.

[KiczalesEtAl'91] Kiczales, G., des Rivières, J. and Bobrow, D. G., The Art of the Metaobject Protocol, MIT Press, 1991.

[KlasEtAl'90] Klas, W., Neuhold, E.J., Schrefl, M., Metaclasses in VODAK and their Application in Database Integration, Arbeitpapiere der GMD, no. 462, 1990.

[LaddagaVeitch'97] Laddaga, R. and Veitch, J., Dynamic Object Technology. In Communications of the ACM 40, 5 (May 1997).

[Lieberherr'94]Lieberherr, K. J., Silva-Lepe, I., Xaio, C. Adaptive Object-Oriented Programming Using Graph-Based Customizations. Communications of the ACM, 1994, 37(5), pp. 94-101.

[Meijler'93] Meijler, T.D. User-level Integration of Data and Operation Resources by means of a Self-descriptive Data Model. Ph.D. thesis, Erasmus University Rotterdam, Sept. 1993.

[MeijlerEtAl'97a] Meijler, T.D., Demeyer, S. and Engel, R., Class Composition in FACE, a Framework Adaptive Composition Environment. In Special Issues in Object-Oriented Programming, Max Muehlhauser (Ed.), Heidelberg: dpunkt, verl. fur digitale Technologie, 1997.

[MeijlerEtAl'97b] Meijler, T.D., Demeyer, S. and Engel, R, Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment. In Proceedings ESEC/FSE '97, M. Jazayeri and H. Schauer (Ed.), LNCS 1301, Springer-Verlag, September, 1997, pp. 94-110.

[Rao'91] Rao, R. Implementational Reflection in Silica. In Proceedings ECOOP'91, P. America (Ed.), LNCS 512, Springer-Verlag, July 1991, pp. 251-267.

[Rieger'97] Rieger, M., Implementing the FACE Object Model in C++, Diploma thesis, University of Berne, June 1997.

[SmithUngar'95] Smith, R.B., Ungar, D., Programming as an Experience: The inspiration for Self. In Proceedings ECOOP'95, W. Olthoff (Ed.), LNCS 952, Springer-Verlag, August 1995, pp. 303-330.

[SteyaertEtAl'96] Steyaert, P., De Hondt, K., Demeyer,S. and Boyen, N., Reflective User Interface Builders. In Advances in Object-Oriented Metalevel Architectures and Reflection, Chris Zimmerman (Ed.), CRC Press - Boca Raton - Florida, 1996, pp. 291-309.

Authors

Serge Demeyer and Matthias Rieger both work for the University of Berne in Switzerland. They can be reached at demeyer@iam.unibe.ch or rieger@iam.unibe.ch and they have their web pages at http://www.iam.unibe.ch/~demeyer/ or http://www.iam.unibe.ch/~rieger/ .

Theo Dirk Meijler works for the Research Department of the Baan development Co. in Ede, The Netherlands. His e-mail is tdmeijler@research.baan.nl.

Edzard Gelsema works for the department of medical informatics the Erasmus University in Rotterdam, The Netherlands. His e-mail is gelsema@mi.fgg.eur.nl and he has a web page at http://www.eur.nl/FGG/MI/people/esg.html.