Because customizing real-world applications is a genuinely complex operation, developers must look beyond the seductive appeal of filling in hot spots in frameworks.

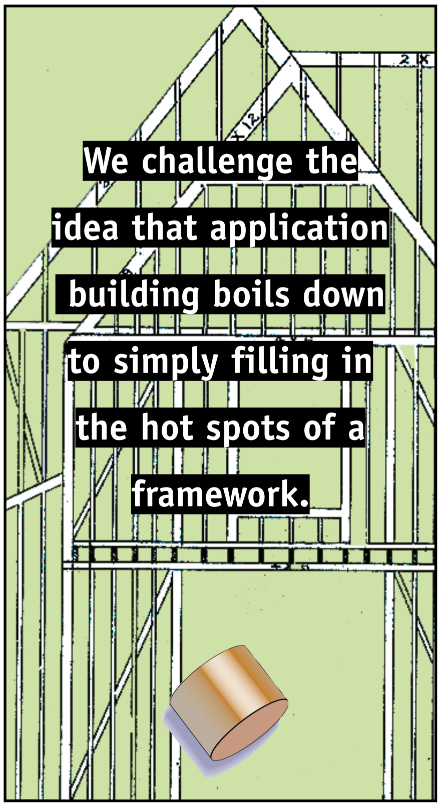
From Custom Applications to Domain-Specific Frameworks

Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen

As a provider of broadcast planning software for television stations, we have to create highly customized software while maintaining the quality standards of an off-the-shelf product. Framework technology plays a strategic role in our business, because there are many similarities in the complex broadcast planning processes of different stations, yet a standard product would satisfy only 70% to 80% of the needs of a typical station. Thanks to frameworks, customers can be offered a standard product that is easily customizable by a small team of software engineers in a cost-effective way. Moreover, the adaptive nature of frameworks provides another advantage; like any other business, television stations need to respond to new and rapidly changing market opportunities. They are faced with rapidly evolving hardware (e.g., digital video broadcasting, distributed video production) and rapid evolution of their products (e.g., interactive TV, electronic program guides). These changes require more malleable software. For broadcasters, the ability to cope with change more efficiently than from similar off-the-shelf applications results in a competitive advantage.

VICTOR SADOWSKI

After several years of incrementally building our framework one project at a time, customizing it for several television stations across Europe, we had considerable experience with framework technology at a technical as well as a managerial level [1]. While the decision to use framework technology is mainly evaluated positively, our experience shows that for many companies like ours, success depends on considerations beyond what can be found in the literature on framework technology. Our goal is not to build frameworks but to provide solutions to customers more efficiently and with higher quality. The details of using a framework as a strategic weapon in attacking a vertical market are largely neglected in the literature; so are the difficulties in evolving a custom application into a framework capturing the domain knowledge of a team of software engineers and domain specialists.


We challenge the idea that application building boils down to simply filling in the hot spots of a framework.

Our use of framework technology goes beyond the idea of selling, buying, and instantiating general-purpose application frameworks. We challenge the state of the art in framework technology. While adhering to the traditional dichotomy between framework engineering and application engineering [3], we take a more evolutionary approach by interleaving both activities, thereby requiring close interaction between framework developers and application engineers. Moreover, we challenge the idea that application building boils down to simply filling in the hot spots of a framework (see Schmid in this issue); in many real-world applications, like ours, customization is far more complex. Here we report on the problems that can be encountered when incrementally building a framework, including the lack of good reuse documentation, version proliferation, poor effort estimation, delta analysis, architectural drift, and overfeaturing. We also point out possible solutions for the future.

History, Goals, Objectives

The initial broadcast planning application was the result of a project for a new Belgian television broadcast station to automate its entire suite of broadcast planning activities. Broadcast planning is a complex process involving many departments (e.g., planning, contract and program, and tape). The resulting groupware application closely reflected the particular interdepartmental work flow and was tightly integrated with existing hardware and software. Although designed and implemented by a six-person team of software engineers lacking experience in broadcast planning, the project remained within budget and on schedule, and ultimately, the users were satisfied. Moreover, due to the use of OO technology, we already delivered a certain degree of adaptability in the application (at the end of the project when the customer changed some basic requirements).

Due to this success and demand from other stations, we began planning to commercialize the application. However, we quickly realized that a standard off-the-shelf product would not satisfy this market, so we formulated the following product goal: Offer a broadcast planning solution that is highly and efficiently customizable to the needs of different television stations and that gives the customer the feeling of a custom system with the qualities of a standard product. Therefore, the system cannot have unnecessary features, it must be adapted to the customer's work process, and it must be integrated with existing hardware and software. Moreover, it should offer the stability and the possibility for future upgrades typically associated with off-the-shelf products.

We also recognized that this goal was not easy to achieve. To install the software at a second broadcast station, the original application was adapted toward the specific needs and infrastructure of that station. After a while, however, the architectures of both applications drifted apart, resulting in severe maintenance problems for our small development team. We also realized that these problems would only become worse. Achieving a higher level of reuse would require a more systematic approach, so we decided to look into framework technology as a possible solution.

Challenging the State of the Art

A framework is usually defined as a skeleton program defining a reusable software architecture in terms of collaboration contracts between (abstract) classes and a set of variation points, or hot spots. The hot spots define where the framework can be customized; collaboration contracts define the rules the customizations must obey. Framework-based development combines two development process models: product development, in the sense that a product (the framework) is offered, and project development, because customers are involved in customizing the product to their specific needs. This duality must be reflected in the process model. More precisely, two activities—framework engineering and application engineering—can be distinguished and require different skills. Framework engineers are responsible for the design and implementation of the framework; application engineers concentrate on the customization of the framework.

A model for framework engineering often described in the literature requires framework development to involve an extensive domain analysis prior to the framework design. In this model, the ultimate goal of framework development is to build—through a small number of iterations—a software architecture that can be turned into a custom application by simply filling in the hot spots. In the context of broadcast planning—and in many other situations—simply filling in hot spots is a delusion. For real-world applications, only a limited number of frameworks can be customized by just filling in the hot spots. In most cases, the customization process is much more complex, sometimes even violating part of the framework architecture. Moreover, the idea of constructing an immutable framework after a limited number of iterations is not realistic. On the one hand, the large upfront investment in the domain analysis and building (mostly) prototype applications for establishing the framework architecture is in most cases not financially justifiable. This analysis is cost-effective only for frameworks that can be sold in a relatively broad market and that have a relatively well-known and stable problem domain (e.g., generic application frameworks). On the other hand, framework developers are confronted by constant changes in the market. As the business evolves, so must the framework. It is simply not possible to conceive a framework that anticipates all future evolutions. A framework is never finished.

A framework's objective is to consolidate the domain knowledge acquired during earlier projects so it can be reused in future projects to realize a product goal. A framework thus constitutes an ever-evolving representation in terms of variations and commonalities of our knowledge of the domain at a given point in time.

In our experience, evolution is called for during any of the following situations:

- New insights in the domain. As more customizations are made, some site-specific concepts may become general concepts and must be incorporated into the framework.
- Complexity of classes. As the framework evolves, classes tend to become more complex. To reduce this complexity, the framework developer has to consider a redesign of the framework that can range from introducing new abstractions (e.g., new abstract classes and the factorization of complex classes) to such advanced refactorings as design patterns [2, 7].
- New design insights. Some design issues are neglected—or even forgotten—in the framework's initial design phase. These issues often turn into problems, possibly affecting performance, during customization of the framework. And the framework often has to be redesigned to solve these problems (e.g., by improving an algorithm).

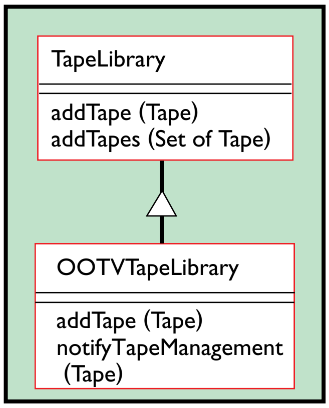


Figure 1. The `TapeLibrary` class hierarchy

Challenges

Therefore, the framework is developed through a stepwise (iterative) construction as we proceed with the customization projects at different television stations. But this method poses the following additional problems concerning evolution and iterative development.

Reuse documentation. In practice, when the goal is reuse of a component or customization of part of a framework, informal "hallway discussions" are needed between the reusers and the original designers to clarify design issues that cannot be extracted from the traditional documentation.

Consider the following simple example. Figure 1 shows a class `TapeLibrary` representing a television station's tape library. The class includes behavior for adding a single tape (`addTape`) and for adding sets of tapes (`addTapes`). Suppose that for a new station OOTV, the tape management department has to be notified each time a tape is added to the library. This notification can be accomplished by subclassing the TapeLibrary class with OOTVTapeLibrary. In order to decide which methods need to be overridden, we need information about which methods depend on which other methods. For example, if we know that addTapes depends on addTape for its implementation, it is enough to override the method addTape to account for notification. If addTapes does not rely on addTape, both methods need to be overridden. The OMT class diagram does not provide sufficient information for this analysis, as it does not state the dependencies between the method implementations. In light of the simplicity of this example, code inspection would work fine. But in practice, inspecting the code to reuse a class is undesirable; this kind of analysis should be feasible at the design level.

The long life span and the strategic role of frameworks means that good design information is essential. Design documentation for frameworks should offer more than what is traditionally offered by OOA/OOD notations. Code inspection is not an option, since it is error prone and time consuming and can lead to customizations that depend too much on the framework's implementation details. Therefore, an intermediate level of description is desirable.

Version proliferation. After each customization, redesign of the framework is considered by the framework engineers. Even after the initial iterations, modifications to the framework still occur. Although these changes are less and less frequent, their consequences are the most difficult to assess. Most such changes have a large impact on the rest of the system and are often incompatible with previous customizations. The scope of the impact makes managing the consistency of the different customizations and the maintenance of the framework itself extremely difficult.

For example, suppose a number of customers complain about the performance of the tape library. A performance profile reveals that the problems are due to the fact that each time a set of tapes is added, a separate database transaction is opened for each tape in the set. Performance could be improved significantly by storing all tapes with a single database transaction. Since the framework designer wants every customer to benefit from the performance gain, the TapeLibrary class has to be redesigned. In the example, it is sufficient to modify the method addTapes so it no longer invokes addTape. This modification leads to inconsistent behavior in OOTVTapeLibrary when the customizer decides to upgrade to the new version of the framework, as the tape management department will not be notified of groupwise-added tapes. Using the terminology of Kiczales and Lamping [5], we conclude that addTapes and addTape have become inconsistent methods (see Figure 2).

Although in the example this inconsistency can be derived from the code, in practice it should be possible to detect these problems without code inspection. In order to estimate the impact of a class modification on its inheritors, more information is needed about the way inheritors reuse their superclasses. Without reuse information on how a customization relies on the framework, it is very difficult to estimate the effort needed to update a customization to a new version of the framework.

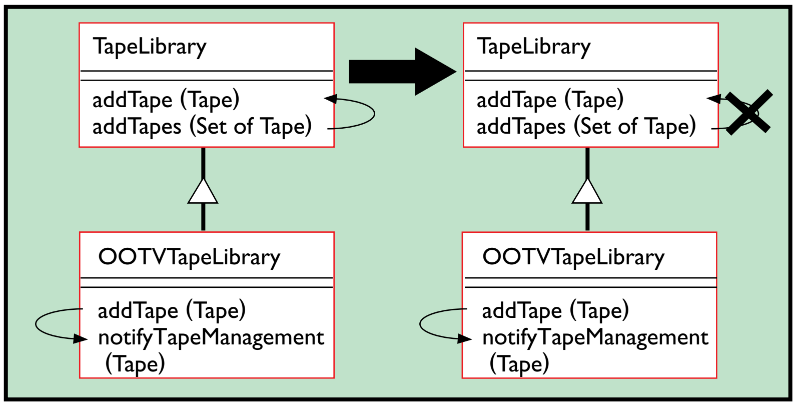


Figure 2. Inconsistent methods

Managing the propagation of changes made to a framework so (re-)users of that framework are not adversely affected remains an essential problem in the development of frameworks. Today, code inspection is the only available strategy for estimating the impact of a class modification. The result is that subtle conflicts with significant consequences are often detected only during the testing phase, if at all.

Delta analysis and effort estimation. Instead of performing an analysis from scratch, application engineers perform a delta analysis. One way to do this is to browse through a prototype with the customer and carefully write down where existing functionality covers the customer's needs, where new functionality is required, and where variations on already existing functionality are necessary. In practice, variations are not always covered by hot spots in the framework. This lack of coverage makes it difficult to assess what parts of the framework can be

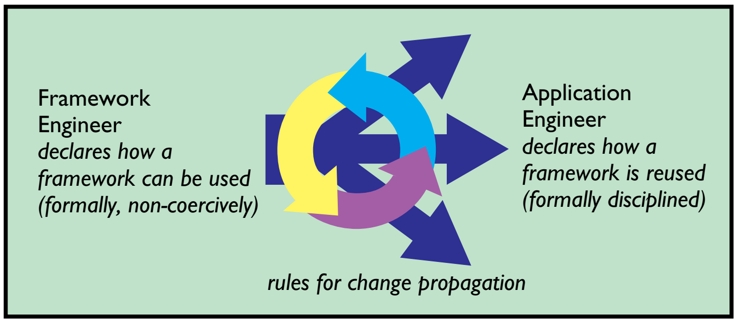

Framework Engineer
declares how a framework can be used (formally, non-coercively)
Application Engineer
declares how a framework is reused (formally disciplined)
rules for change propagation

Figure 3. Contract between framework and application engineers

reused, what parts need to be adapted before reuse, and how much effort is required by the adaptation. Without proper documentation, the application engineer is compelled to retrieve the needed information from the framework engineer through informal communications. But such communications create a large overhead. Our experience is that application engineers are often forced to inspect the source code, thereby short-cutting the framework developer. Short-cutting is a serious impediment to the iterative development process. The framework developer needs insight into how the framework is actually reused to be able to improve the reusability of the framework. A more formal notation allowing application developers to express how the framework needs to be customized for a particular customer. How it is actually customized remains a major challenge.

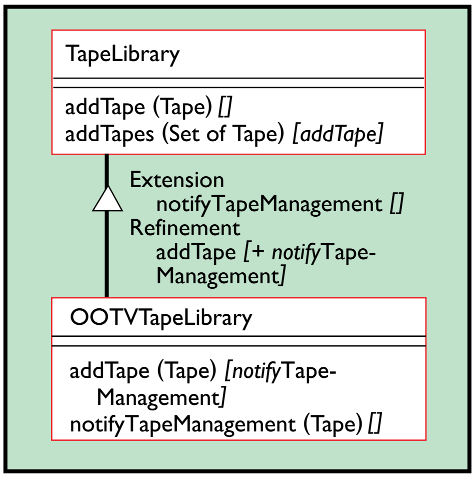

TapeLibrary
addTape (Tape) *[]*
addTapes (Set of Tape) *[addTape]*
Extension
notifyTapeManagement *[]*
Refinement
addTape *[+ notify*Tape-Management*]*
OOTVTapeLibrary
addTape (Tape) *[notify*Tape-Management*]*
notifyTapeManagement (Tape) *[]*

Figure 4. Reuse contract notation for refinement of a class

Architectural drift. Defining a mature framework architecture is one achievement; actually enforcing its use is another. Ignorance, deadline pressure, and the not-invented-here syndrome (see Schmidt and Fayad in this issue) often result in solutions that are either reinventions of designs already present in the framework or solutions that needlessly break the framework architecture. Enforcing the architecture of a framework without overconstraining the customizer is a necessity.

Reuse documentation in the form of cookbooks is too constraining, because it documents only a limited set of predefined ways of reusing a framework. When reuse documentation is too limited, opportunities for reuse in many cases are not spotted, because reusers have no notion of what is available for reuse and how a component can be reused. Current OOA/OOD notations do not solve the problem either; they are not equipped to document how a class can be reused or even to express how to reuse a framework in a disciplined fashion—without violating its architectural design. The lack of documentation for disciplined reuse has implications for the quality of the resulting software. Improper reuse of a component often results in bugs or incompatibilities at a later stage of software development.

Overfeaturing. Developers tend to migrate features to the framework kernel in order to reduce customization efforts. Migrating features results in overfeaturing, so applications containing features not relevant for a particular user are still part of the "standard package." Overfeaturing makes the framework more expensive, more complicated, and less reusable for future customers. Great care must be taken when deciding which features to add to the framework. Walking the thin line between standard product development and project-based development, we set up a User Advisory Board that acts as a discussion forum for our strategic customers, helping decide which features are to be integrated into the framework.

Reuse Contracts

The key to successful software development with frameworks is good cooperation between framework engineers and application engineers. Framework engineers have the important advisory role of giving application engineers techniques for increasing the generic aspects of their designs and code, advice on how to avoid architectural drift during the customization process, and advice on how to reuse the framework. Application engineers have to give feedback about framework customization so it can be improved. We conclude that this cooperation should be based on an explicit reuse contract [8] (see Figure 3).

Since framework reuse boils down to the reuse of single (abstract) classes and the reuse of the interaction between classes, reuse contracts have been defined for these two forms of reuse. Single-class reuse contracts are based on Lamping's notion of specialization inter-

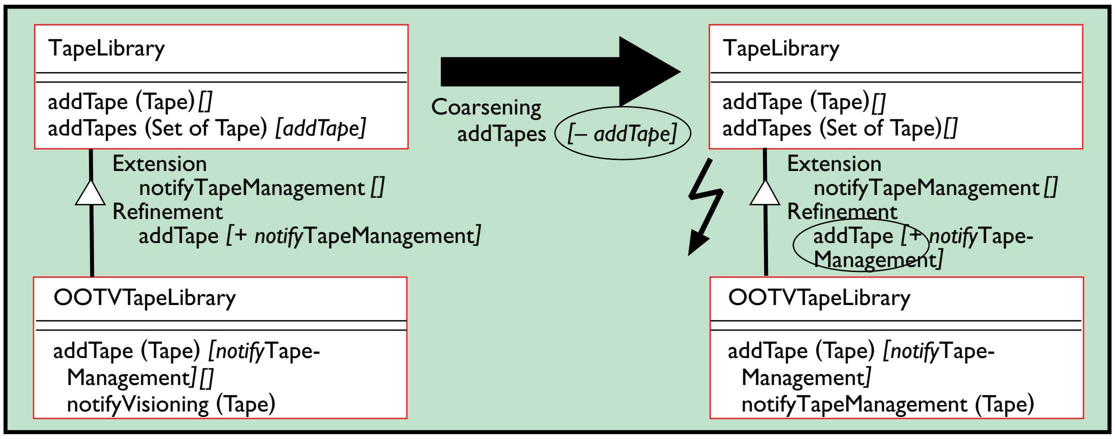

TapeLibrary
addTape (Tape) []
addTapes (Set of Tape) [addTape]
Extension
notifyTapeManagement []
Refinement
addTape [+ notifyTapeManagement]
OOTVTapeLibrary
addTape (Tape) [notifyTape-Management] []
notifyVisioning (Tape)
Coarsening
addTapes [– addTape]
TapeLibrary
addTape (Tape) []
addTapes (Set of Tape) []
Extension
notifyTapeManagement []
Refinement
addTape [+ notifyTape-Management]
OOTVTapeLibrary
addTape (Tape) [notifyTape-Management]
notifyTapeManagement (Tape)

Figure 5. Conflict detection based on reuse operators

faces [8]. Multiclass reuse contracts are based on Helm and Holland's contracts [4]. Reuse contracts focus on how exactly to apply these ideas in practice, as shown in Figure 3.

Here we focus on single-class reuse contracts (see Figure 4). For each class the reuse contract lists the method signatures of the methods that are relevant to the design of the framework, the associated specialization clauses (in italics in Figure 4), and an annotation for abstract methods (with an "abstract" keyword not in Figure 4 because all the methods in it are concrete). The specialization clauses [6] list the names of the methods invoked through self-sends. Although specialization clauses provide only static information (information corresponding directly to the source code), our experience reveals that in practice this information is enough to determine which methods can be inherited and which methods must be overridden.

Reuse operators document how classes are derived through inheritance. Reuse operators provide more information than plain class inheritance by encoding the various ways a class is reused (see the annotations next to the triangle in Figure 4). Refinement, extension, and concretization are design-preserving operators. Their respective inverses—coarsening, cancellation, abstraction—are design-breaching operators. Refinement refines the design of methods by extending the specialization clause; extension adds new methods; and concretization makes abstract methods concrete. Although these are not the only oper-

Table 1. Correlating reuse operators O_1 and O_2 for conflicts

O_1 / O_2	extension n	refinement n[...]	refinement m[n]	cancellation n	coarsening n[...]	coarsening m[–n]
Extension n	signature collision	—	—	—	—	—
refinement n[...]		specialization clause conflict	method capture	dangling reference	specialization clause conflict	inconsistent methods
refinement m[n]		method capture	specialization clause conflict	dangling reference	method capture	specialization clause conflict
cancellation n		dangling reference	dangling reference	signature collision	dangling reference	dangling reference
coarsening n[...]		specialization clause conflict	method capture	dangling reference	specialization clause conflict	inconsistent methods
coarsening m[–n]		inconsistent methods	specialization clause conflict	dangling reference	inconsistent methods	specialization clause conflict

Reuse operators

Refinement m[n]: Add an invocation of n to the specialization clause of m.
Extension n: Add a new method n to the client interface.
Coarsening m[–n]: Remove an invocation of n from the specialization clause of m.
Cancellation n: Remove an existing method n from the client interface.

Conflicts

Signature collision: Both reuse operators introduce or remove the same method signature.
Specialization clause conflict: Both reuse operators change the specialization clause of the same method.
Dangling reference: One reuse operator refers to a method while the other one removes it.
Method capture: One reuse operator changes the specialization clause of a method under the assumption that the method is not involved, while the other reuse operator adds the method invocation.
Inconsistent methods: One reuse operator changes the specialization clause of a method under the assumption that the method is invoked, while the other reuse operator removes the method invocation.

ations imaginable, they do coincide with the typical ways abstract classes are reused.

Single-class reuse contracts document how a class can be reused. Specialization clauses help determine which methods need to be overridden. For example, the application engineer for OOTV decided that overriding the method `addTape` was enough to achieve the desired behavior based on the specialization clause of `addTapes` (see Figure 4).

Reuse operators document how a class is actually reused. In the TapeLibrary example, the inheritance association for the `OOTVTapeLibrary` is annotated with an extension and a refinement reuse operator to express that the `notifyTapeManagement` method was added and that the design of the addTape method is refined (`notifyTapeManagement` is added to the specialization clause). Besides documenting how a class is reused by inheritors, reuse operators can also be used to document the evolution of a parent class (see top of Figure 5). Conflict detection and effort estimation can then be performed automatically by correlating the reuse contracts that document the evolution of a parent class with the reuse contracts that document inheritors.

Table 1 shows the conflicts that can occur in an inheritor derived by a reuse operator O_2 when the parent class is changed by applying a reuse operator O_1.

In the redesign of the `TapeLibrary` class in Figure 2, the fact that `addTape` and `addTapes` have become inconsistent in class `OOTVTapeLibrary` can now be derived directly from the reuse contracts. `OOTVTapeLibrary` refines a method that has been removed from the specialization clause through a coarsening when changing from the old parent class to the new parent class.

Simple formal rules were defined to signal possible conflicts in existing inheritors when changes are made to their parent classes [8]. Most of the possible conflicts can be expressed in terms of reuse contracts and reuse operators rather than at the level of interfaces and calling structures. This level of expression allows developers to reason about change in more intuitive terms and on a higher level. Moreover, these rules are the basis for developing tools that can automatically assess the impact of changes made to the framework, forecast which conflicts might occur and when, and guide application developers in understanding both where testing is needed and how to fix the conflict.

Conclusions

The issues in constructing frameworks for developing applications in a well-known and stable problem domain (justifying an up-front investment in light of a large potential market) are relatively well documented. However, in many other cases, like those in which framework technology provides more quality to the customer, current technology is insufficient. In particular, the problems related to evolutionary development, such as architectural drift and version proliferation, remain a challenge. Moreover, the prevalent model of application building (filling in hot spots) is too naive for most real-world applications. Better support is needed to document how to reuse frameworks and how to perform effort estimation. We point to the reuse contracts we developed in a joint research project as an indication of how existing notations can be enhanced for this purpose. C

References

1. Codenie, W., Verachtert, W., and Vercammen, A. PSI: From custom-developed application to domain-specific framework. In the *Addendum to the Proceedings of OOPSLA'96* (San Jose, Calif., Oct. 6–10). ACM/SIGPLAN, New York, 1997.
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.
3. Goldberg, A., and Rubin, K. *Succeeding with Objects: Decision Frameworks for Project Management*. Addison-Wesley, Reading, Mass., 1995.
4. Helm, R., and Holland, I.M. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP '90* (Ottawa, Canada, Oct. 21–25). ACM/SIGPLAN, New York, 1990, pp. 169–180.
5. Kiczales, G., and Lamping, J. Issues in the Design and Specification of Class Libraries. In *Proceedings of OOPSLA '92* (Vancouver, British Columbia, Canada, Oct. 18–22). ACM/SIGPLAN, New York, 1992, 435–451.
6. Lamping, J. Typing the specialization interface. In *Proceedings of OOPSLA '93* (Washington, D.C., Sept. 26–Oct. 1). ACM/SIGPLAN, New York, 1993, pp. 201–214.
7. Opdyke, W.F., and Johnson, R.E. Creating abstract superclasses by refactoring. In *Proceedings of CSC '93* (Indianapolis, Feb. 16–19), ACM Press, New York, 1993, pp. 66–72.
8. Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96* (San Jose, Calif., Oct. 6–10). ACM/SIGPLAN, New York, 1996, pp. 268–285.

Wim Codenie (winn@voopartners.com) is a framework engineer at OO Partners/MediaGenIX in Strombeek-Bever, Belgium.
Koen De Hondt (kdehondt@vub.ac.be) is a researcher in the Programming Technology Lab of the Vrije Universiteit Brussel in Belgium.
Patrick Steyaert (prsteyae@vub.ac.be) is a post-doctoral researcher in the Programming Technology Lab of the Vrije Universiteit Brussel in Belgium.
Arlette Vercammen (arlette@oopartners.com) is managing director of OO Partners/MediaGenIX.