Vrije Universiteit Brussel
Faculteit Wetenschappen

Evolving Custom-Made Applications into Domain-Specific Frameworks

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

Techreport vub-prog-tr-97-03

Programming Technology Lab
Computer Science Department
PROG/DINF/WE
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3525
Tel: (+32) 2-629-3308
ftp://progftp.vub.ac.be/
http://progwww.vub.ac.be/

-- SUBMISSION --
Submitted to the CACM, October 1997, Issue on "Object-Oriented Application Frameworks"

Evolving Custom-Made Applications into Domain-Specific Frameworks

Wim Codenie*, Koen De Hondt†, Patrick Steyaert†, Arlette Vercammen*

*OO Partners[1]
†Programming Technology Lab, Vrije Universiteit Brussel[2]

Abstract. Framework development involves many challenges, from a technical as well as a managerial point of view: *reuse documentation, proliferation of versions, effort estimation, delta analysis, architectural drift*. Several of these challenges are not, or scarcely, addressed by current OOA/OOD methodologies nor by the framework literature. We elaborate on the experiences we had with building a framework for developing commercial applications in the market of broadcast planning. In particular it is shown how we dealt with these challenges during the development of the framework. Reuse contracts are put forward as an indicative solution to a number of problems we encountered.

1. Introduction

OO Partners is a provider of broadcast planning software tailored to the specific needs of television stations. Framework technology was chosen because of the similarities between the needs of the different customers. While broadcast planning is a fairly complex process many similarities exist between the workflow of different television stations. Yet a standard product would only satisfy 70% to 80% of the needs of a typical customer. Thanks to the usage of frameworks, customers can be offered a standard product that is easily customizable by a small team of software engineers in a *cost-effective* way. With this approach OO Partners distinguishes itself from its competitors that provide off-the-shelf solutions that never result in the same degree of customer satisfaction. Moreover, the *adaptive nature* of frameworks provides another advantage. As in any other business, television stations use software more

[1] *Authors' addresses*: OO Partners bvba. Otto de Mentockplein 19, B-1853 Strombeek-bever, Belgium. *Tel*: +32 2 263 04 00, *Fax*: +32 2 267 71 51, *E-mail*: {wim, arlette}@oopartners.com

[2] *Authors' addresses*: Programming Technology Lab, Vrije Universiteit Brussel. Pleinlaan 2, B-1050 Brussel, Belgium. *Tel*: +32 2 629 34 92, *Fax*: +32 2 629 35 25, *E-mail*: {kdehondt, prsteyae)@vnet3.vub.ac.be

and more in a strategic way, for instance to respond to new market opportunities. This requires a higher degree of malleability of the software, which in our case was realized with framework technology. Several of our customers already benefited from this malleability. Examples are the integration of the software with new transmission hardware and the incorporation of new kinds of video media in the application. Undoubtedly, the ability to cope with change more efficiently than similar off-the-shelf applications on the market results in a *competitive advantage.* Customers tend to choose for an open solution to their problems.

Frameworks are strategic in our business strategy, and the decision to use them is mainly evaluated positively. However, it is our experience that, for many companies like ours, success with this technology depends on a number of considerations that are 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 more quality. The particularities of using a framework as a strategic, albeit supportive, weapon in attacking a vertical market are largely neglected. So are the difficulties in evolving a custom-made application into a framework that captures the domain knowledge of a team of software engineers and domain specialists. After several years of incrementally building our framework on a project-to-project basis, and customizing it for several television stations across Europe, we gathered considerable experience in applying framework technology, at a technical as well as a managerial level (see also [1]). We believe that other kinds of framework development can benefit from these experiences. In this paper we report on the problems we encountered and on our experiences with the usage and development of frameworks in custom-developed software for a particular vertical market. We also point out possible solutions for the future.

2. Evolution of the Process Model

2.1 From Conventional Model to Framework Process Model

Since our start in the television business, our software development process model has evolved from a traditional project-based model to a framework-based development model in which systematic reuse across multiple projects plays a central role.

The initial application, called PSI, was custom-developed for a specific television station. It was the result of a project carried out for a new Belgian television broadcast station to automate their broadcast planning process. The application was designed and implemented by a small team (6 persons) of skilled software engineers that had no previous experience in the domain of broadcast planning. Moreover, because the television station was founded just before the start of the project, not all the requirements were exactly known in advance. To cope with this, an incremental and iterative approach based on prototyping was adopted. At this time this was the major reason why an object-oriented solution (Smalltalk) was chosen. The development team was divided in subteams, based on an enterprise-wide analysis [3]. Each team member was involved in the analysis, design and implementation phase of at least one subsystem. This significantly reduced the communication overhead between these phases, and helped each team member in acquiring considerable domain knowledge.

During the installation of the initial planning application other broadcast stations showed an interest in PSI. However, implementing the software at another television station could not be realized by a simple duplication of the original. Although the bulk of the application could be reused, the differences between the planning processes of different broadcast stations could not be neglected. So, to install the software at a second broadcast station, the original application needed to be adapted towards the specific needs of that station. However, it soon became apparent that this was not the most efficient way of working. After a while, the architectures of both applications drifted apart resulting in severe maintenance problems for OO Partners' small development team. We also had problems concerning reuse, something that was not that much of an issue until then. In particular, problems were encountered with porting features (added after the split) from the second application back into the original application.

It was in this period that we began to explore new market opportunities for PSI. We realized, however, that the above mentioned versioning problems would only become worse. In order to achieve a higher level of reuse, a more systematic approach was necessary. In our case this was realized by the construction of a framework for television broadcast planning. Our goal in building this framework is summarized below.

> ***PSI goal**: Offer a broadcast planning solution that is highly, and efficiently customizable to the needs of different television stations. The solution must give the customer the feeling of a custom-built system with the qualities of a standard product. So, it should not contain needless features, must be adapted to the customer's work process, and must be integrated with existing hardware and software. Moreover it should offer the stability and the possibility for future upgrades that is typically associated with off-the-shelf products.*

Obviously this had an impact on our process model.

2.2 Framework Objective

A model for framework construction often encountered in the literature is that framework development requires an extensive domain analysis prior to the framework design and that the ultimate goal of framework development is to build — through a small number of iterations — a software architecture that can be turned into a customized application by simply filling in the hot spots. In the context of the PSI framework — and in many other situations — this is a delusion. For real-world applications, those parts of the applications that can by customized by just filling in the hot-spots are rather limited. 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 up-front investment in performing the domain analysis and building (mostly) prototype applications for establishing the framework architecture is in most cases not feasible from an economical point of view. This is only cost-effective for frameworks that can be sold in a relatively broad market and that have a relatively well-known and stable problem domain (such as is the case with most generic application frameworks). On the other hand, framework developers are confronted with the never abating change on the market: As the business evolves, so must the framework. It is simply not possible to conceive a framework that anticipates all future evolutions. Therefore a framework is never finished.

> ***PSI framework objective****: Consolidate the domain knowledge acquired during previous projects in a framework so that it can be reused in future projects as to realize the PSI goal. The framework thus constitutes an ever-evolving representation, in terms of variations and commonalities, of our knowledge of the domain at a certain point in time.*

According to this objective the framework is developed by a step-wise (iterative) construction as we go along with the customization projects at different television stations. This obviously poses an additional range of problems concerning *evolution* and *iterative development*. After each customization, redesigns of the framework are considered. These redesigns can range from introducing new abstractions (introduction of new abstract classes) and factorizations [5] (e.g. making complex classes more reusable) up to more advanced refactorings such as applying design patterns [2]. Even after the initial iterations, modifications to the framework still occur. Although these latter changes occur less frequently, their consequences are the hardest to assess. Most often such changes have a large impact on the rest of the system and are often incompatible with previous customizations. This makes the management of the consistency between the different customizations and the maintenance of the framework itself extremely difficult.

Another problem that arises during step-wise development of a framework is overfeaturing, i.e. end-user applications that contain features that are not relevant for this particular end-user but are part of the "standard package". In practice this poses additional managerial problems. A tendency exists to migrate features to the framework kernel in order to reduce customization efforts. However, this makes the framework more expensive, more complicated, and less reusable for future customers. Great care must be taken in what features are added to the framework and what not. To walk the thin line between standard product development and project based development, OO Partners has set up a PSI User Advisory Board that acts as a discussion forum for strategic customers. The user advisory board decides which features are integrated in the framework.

2.3 Framework Process Model

Framework-based development is a combination of two development process models. It is product development in the sense that a product (the framework) is offered. At the same time it is also project development because the customer is involved in customizing the product to his/her specific needs. This duality must be reflected in the process model. More precise, two activities can be distinguished that require different skills: *framework engineering* and *application engineering*. The framework engineers are responsible for the design and implementation of the framework (i.e. domain analysis, building reusable designs, refactorings, ...). Their activities typically span multiple projects. The application engineers concentrate on the customizations of the framework (i.e. delta analysis, reusing designs, filling in or refining hot spots in the framework, ...). Framework development proceeds largely in parallel with application development. Today, at OO Partners, 70% of the development activities can be classified as application engineering while 30% can be considered framework engineering.

The application engineering activities differ from traditional project development activities in that they are not aimed at building an application from scratch, but rather concentrate on customizing an existing standard product (the framework). Even during the earliest phase of a project — the tender submission phase — we are confronted with the particularities of this approach. For example, to be able to make a proper estimation of the costs involved in customizing the framework to the needs of a particular television station, a profound knowledge of the framework is indispensable. Today, we realize this by closely involving the framework engineers in this phase as they can provide the most detailed information about the feasibility and efforts needed to implement extensions and modifications. The framework approach also has an impact on the requirements analysis. Instead of performing an analysis from scratch, the application engineers use a prototypical PSI application to capture the user requirements. This is a highly interactive process. Together with the customer, the application engineers browse through the prototype and carefully write down all the differences with the prototype. This can be either entirely new functionality or variations on already existing functionality. The output is a document, the *delta analysis*, that serves as input for the design and implementation phases.

In the case of OO Partners, framework engineering consists of certification activities and advising activities. Certification activities are done on a regular base. An example is bringing analysis documents up to date with new versions of the framework so that future projects can benefit from these new versions. Another example are refactorings. If necessary, existing designs and implementations are refactored by the framework engineers to make them more reusable. Special attention must be given to the correct assessment of the impact of refactorings on existing customizations. A final example of a certification activity is the consolidation of domain knowledge by making abstractions across multiple customizations of the framework. Framework engineers also have an important advising role. For example, they impart application engineers with techniques to increase the genericity of designs and code, and try to avoid architectural drift during the customization process as explained in section 4.2.

A lot of communication takes place between framework engineers and application engineers. In OO Partners' case, the communication overhead is reduced by involving — to a certain extent — framework engineers in the application engineering process and vice versa. This approach is not ideal as it only works for small development teams. Furthermore, both activities require different skills that are seldom mastered by a single software engineer. Framework engineers tend to have more technical skills, while application engineers are more interested in domain issues of television planning. As we will explain in the next sections, a good communication between the two is the key to successful software development with frameworks.

3. Challenges for Framework Engineers

Today we observe that there is too much verbal — and therefore informal — communication between application engineers and framework engineers. Establishing a more formal communication channel is a necessity. Ultimately such communication must be based on a formal notation that goes beyond current state-of-the-art OOA/OOD notations (OMT, UML, ...). Current notations only target the transition between the conventional project phases (analysis, design, implementation). Frameworks introduce an extra dimension in the process model. Traditional notations fall short in supporting this extra dimension. Sections 3 and 4 review the problems that arise due to this extra dimension. Section 5 gives an indicative solution to these problems.

3.1 Reuse Documentation

The long life-span of a framework makes that the number of people involved in its development is usually much higher than in traditional development. Providing good design information is therefore essential. This need is amplified by the strategic role that frameworks play for the business. Design documentation for frameworks should offer more than what is traditionally offered by OOA/OOD methodologies (e.g. OMT class diagrams). What we often see in practice is that to reuse a component or to customize a part of the framework, an informal "hallway discussion" is needed between the reusers and the original designers. The purpose of this discussion is to clarify design issues that cannot be extracted from the traditional documentation. Hairy details about the internal functionality of components are often not documented and restrict the opportunities for reuse. To lever the reuse process, reuse documentation should be more targeted toward answering the tough questions involving reuse: How can components be reused? How are components reused? How are changes in the framework propagated to the customizations?

Current OOA/OOD notations lack even the most elementary notation for documenting how a single class can be reused. Consider the following simple example. Figure 1 shows a class TapeLibrary that represents a tape library of a television station. The class includes behavior to add a single tape (addTape) and to add sets of tapes (addTapes). Both the implementation and an OMT class diagram are given.

```
Class TapeLibrary
   method addTape (aTape: Tape) is
      ...code to add aTape...
   method addTapes (aTapeSet: Set of Tapes) is
      for tape in aTapeSet do
            self.addTape(tape)
end
```

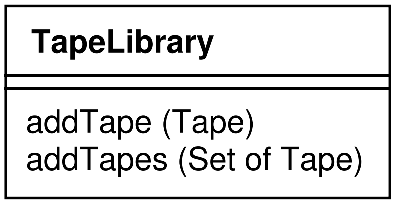

Figure 1. The TapeLibrary class

Suppose that for a new television station OOTV, the tape visioning department must be notified each time a tape is added to the library. This can be realized by customizing the framework, in particular by subclassing the TapeLibrary class with OOTVTapeLibrary. In order to decide which methods need to be overridden, we need information on which methods depend on what other methods. For example, if we know that addTapes depends on addTape in its implementation, it is sufficient to override the method addTape to take notification into account. If on the other hand 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. Because of the simplicity of the example, code inspection works fine here. In practice inspecting the code to reuse a class is undesirable. This kind of analysis should be feasible at the design level. An intermediate level of description is needed.

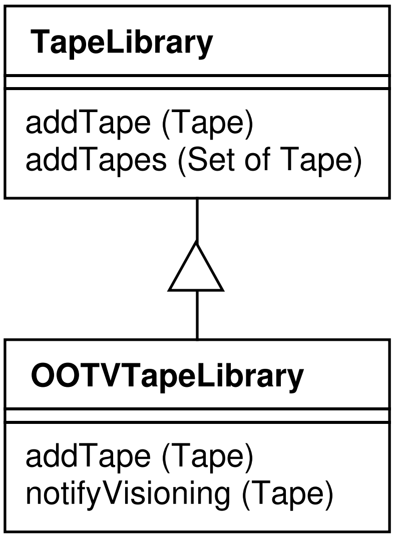

Figure 2. Subclass of TapeLibrary

From the example it is clear that conventional notations need to be enhanced with information for reusers. If even in the simplest imaginable form of reuse, i.e. reusing a single class, good notational support for reuse information is not available, it is certainly lacking for the more complex forms of reuse such as reuse of co-operating classes and frameworks.

3.2 Proliferation of versions

It was already mentioned that evolution of the framework is an instrument to acquire domain knowledge and to reduce up-front investments. In our experience evolution is called for when one of the following situations arises.

- 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.
- Classes become too complex: 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.
- New design insights: improved algorithms, better performance, ...

For example, suppose a number of customers complains 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 must be redesigned. In the example, it suffices to modify the method addTapes so that it does no longer invokes addTape. This leads to inconsistent behavior in OOTVTapeLibrary when the customizer decides to upgrade to this new version of the framework as the visioning department will not be notified of groupwise added tapes. Using the terminology of Kiczales and Lamping [4] we say that addTapes and addTape have become inconsistent methods.

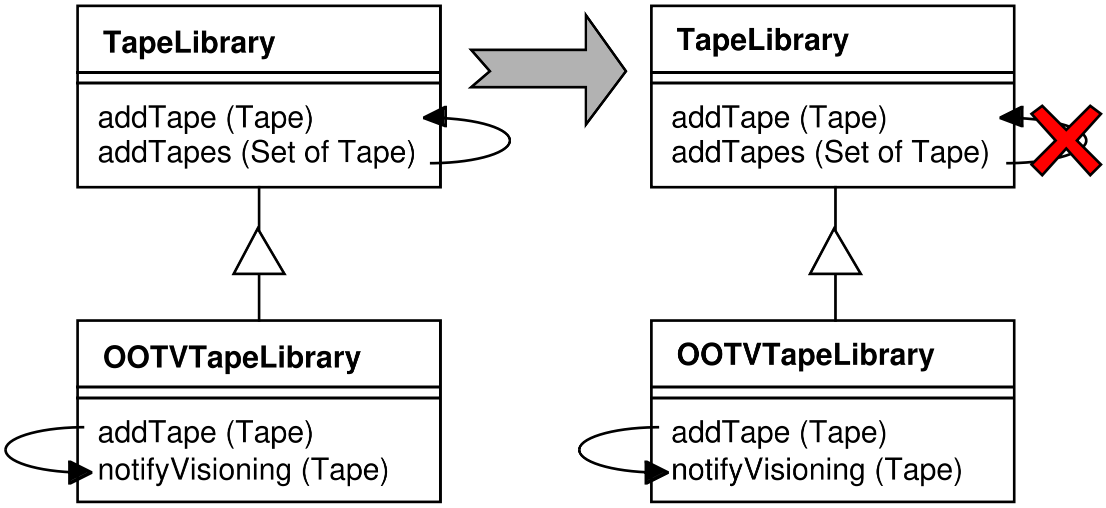

Figure 3. Inconsistent methods

Although in this simple 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 hard to estimate the effort needed to update that customization to a new version of the framework.

Managing the propagation of changes made to a framework so that (re-)users of that framework are not invalidated remains an essential problem in the development of frameworks. Today, the only available strategy to estimate the impact of a class modification is code inspection. The result is that subtle conflicts with important consequences are often only detected during the testing phase, if at all. Because conflicts upon change cannot be easily detected, it is difficult to predict the work effort to update existing applications. Because the lack of documentation on what aspects possible reusers can rely on, it is difficult for developers of a reuse library to decide whether making a certain change to the reuse library is a good idea or not. In a similar vein, the lack of good notations makes it hard for application developers to understand where testing is needed when a framework has changed and how to fix problems when they occur.

4. Challenges for Application Engineers

4.1 Delta Analysis and Effort Estimation

Effort estimation is, for economical reasons, essential in framework development in order to assess the cost of a customization or a redesign of a framework. Based on the delta analysis, the application engineer needs to assess the required changes and needs to estimate how much work it will take to customize the framework to the specific needs of a customer. He wants answers to questions such as: Which classes have to be subclassed? Which methods can be inherited? Which methods have to be added/overridden? How much will this cost?

In our case, framework engineers have documented the framework using OMT class diagrams. Although this documentation can be sufficient for application engineers to get an initial insight in the components of the framework, it does not (cannot) provide the desired information. As a result, the application engineer is compelled to retrieve the needed information from the framework engineer through informal communications. It is needless to say that such informal discussions create a large overhead. Our experience is that application engineers are often compelled to inspect the source code, thereby short-cutting the framework developer. This in its turn is a serious impediment to the iterative development process. The framework developer needs to gain insight in how the framework is actually reused in order to be able to improve the reusability of the framework. A more formal notation that allows application developers to express how the framework needs to be customized for a particular customer, and how it is actually customized remains a major challenge. Framework engineers can profit from this information to ameliorate the framework. Without such notation it is, for example, impossible to trace why a framework was redesigned in a certain way, or to gain insights in how a certain application differs from the standard framework without code inspection.

4.2 Architectural drift

A potential problem during application development is architectural drift. Defining a mature framework architecture is one achievement; enforcing its use is another. Ignorance and deadline pressure often result in solutions that are either re-inventions of designs already present in the framework, or solutions that needlessly break the framework architecture. Enforcing the architecture of a framework without over-constraining the customizer is a necessity.

Reuse documentation in the form of cookbooks is too coercive because it only documents a limited set of pre-defined ways of reusing a framework. On the other hand, current OOA/OOD notations are not satisfactory either. As explained before they are not equipped to document how a class can be reused, let alone to express how to reuse a framework in a *disciplined* fashion — i.e. without violating its architectural design.

When reuse documentation is too coercive, in many cases opportunities for reuse are not spotted because reusers have no notion of what is available for reuse and how a component can be reused. The lack of documentation for disciplined reuse has implications on the quality of the resulting software. A component that is not reused properly often results in bugs or in incompatibilities at a later stage during software development.

5. Co-operation Support for Application and Framework Engineers

5.1 Reuse Contracts

In a joint research project the Programming Technology Lab and OO Partners have been investigating the problems of the previous section and how existing methodologies must be enhanced in order to solve them. Stepping back a bit, and looking at the problems that can be observed when practicing framework reuse — lack of support for change propagation, lack of feedback for iterative design, etc. — one can only conclude that good reuse requires a strong co-operation between framework providers and framework reusers. Therefore, our conjecture is that the co-operation between the framework engineer and the application engineer should be based on an *explicit reuse contract* (see Figure 4). The framework engineer must document that part of the design of the framework that is relevant to application engineers, while these latter must declare how the framework is actually reused. As argued above, the contract should allow disciplined reuse of the framework without being too coercive and be formal enough to allow the management of changes. When framework reuse is based on such a formal contract, application engineers can profit from the changes made to the framework because formal rules can be defined that help in assessing the impact of changes. Framework engineers can profit from the information given by reusers to make their framework more reusable. This contributes to the solution of both the *proliferation of versions* problem and the *iterative development* problem.

Figure 4: Contract between framework and application engineer

The notion of reuse contracts is very general and is applicable to many forms of reuse. Reuse contracts come in different flavours depending on the development phase in which reuse takes place, and on what is reused: reusing single classes by inheritance, reusing multiple classes by late-binding polymorphism. For example, in [6], Steyaert et al. introduce reuse contracts for (abstract) classes to provide an explicit representation of the design decisions taken to build an abstract class.

5.2 Example

A reuse contract (as depicted in Figure 5) for classes includes information such as: which methods are part of the client interface of the class, which methods are invoked by other methods through self sends (the so-called specialization clause, in italics in the figure), and which methods are abstract and which are concrete (with an "abstract" keyword, not shown in the figure because all methods are concrete). Only information relevant to the design is included; auxiliary or implementation-specific methods are not mentioned in a reuse contract. An application engineer who wants to customize a framework, employs the corresponding reuse contracts to determine which classes need to be subclassed and which methods need to be overridden. For a concretization of a framework these classes and methods are easily detected, since the abstractness property of a method is explicitly recorded in a reuse contract. For other customizations, spotting these classes and methods is driven by inspection of other annotations of methods, such as specialization clauses. Thus reuse contracts enable the application developer to spot candidates for customization on the basis of more precise information than is provided by OMT class diagrams or source code.

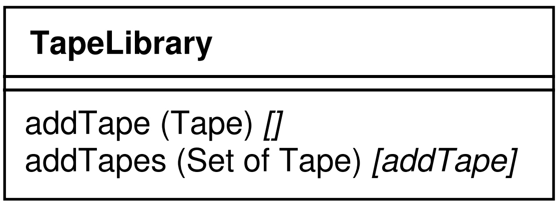

Figure 5. Reuse contract notation for the TapeLibrary class

Although reuse contracts only provide syntactic information, it is our experience that in practice this is enough to determine which methods can be inherited and which methods must be overridden. In our example, by looking at the specialization clauses of the methods in the reuse contract for the TapeLibrary class, the application engineer for OOTV decides that overriding the method addTape is enough to achieve the desired behavior, since addTapes invokes addTape (see Figure 6).

Reuse contracts can only be manipulated by means of reuse operators that provide more information than plain class inheritance (see the refinement annotation in Figure 6). They do so by encoding the different ways a class is reused. Refinement, extension and concretization are design preserving operators. Their respective inverses, coarsening, cancellation and abstraction, are design breaching operators. Refinement refines the design of methods, extension adds new methods, and concretization makes abstract methods concrete. Although not the only operations imaginable, they do coincide with the typical ways to reuse abstract classes. In our TapeLibrary example, the inheritance association for the OOTVTapeLibrary is annotated with the refinement reuse operator to express that the design of the addTape method is refined (a self send is added).

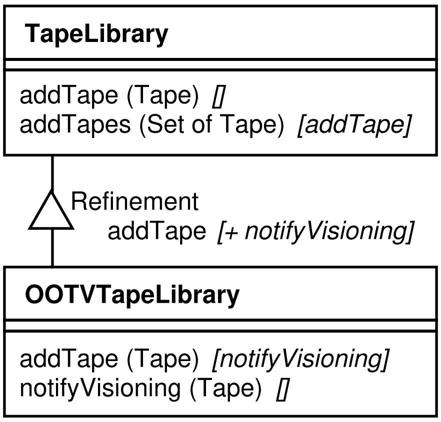

Figure 6. Reuse contract notation for refinement of a class

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 7). 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. In the redesign of our example in section 0, 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 by a coarsening, when changing from the old parent class to the new parent class.

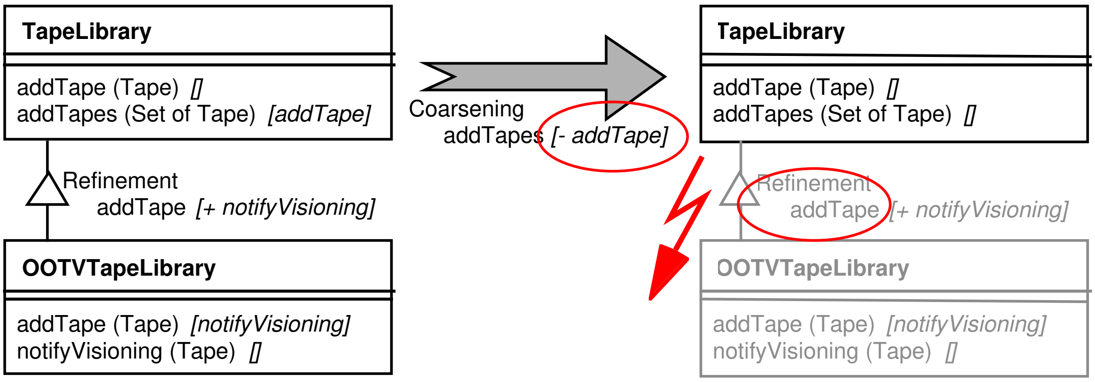

Figure 7. Conflict detection based on reuse operators

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

6. Conclusion

The issues involved in the construction of frameworks for developing applications in a well-known and stable problem domain, and in which an up-front investment is justified because of a large potential market, are relatively well-documented. Contrary to this, we report on our experiences with the construction of a framework, that evolved from a custom-made application and was developed step-wise along with different customization projects at different customer sites. Obviously, this puts a greater emphasis on evolutionary (incremental and iterative) development and its challenges. We reported on the problems we encountered in practice — effort estimation for customizing the framework, delta analysis, proliferation of versions, architectural drift — and concluded that even though framework construction and application development require different skills, good communication between framework engineer and application engineer is essential for success. However, this communication is insufficiently supported by currently available tools and notations, if supported at all. Reuse contracts, that we have developed in a joint research project, are given as an example of how existing notations can be enhanced for this purpose.

7. References

[1] Codenie, W. and Verachtert, W. and Vercammen, A. PSI: From Custom Developed Application to Domain-Specific Framework. To appear in *Addendum to the Proceedings of OOPSLA'96* (Oct. 6-10, San Jose, California). ACM/SIGPLAN, New York, 1997.

[2] Gamma, E. and Helm, R. and Johnson, R. and Vlissides, T. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1994.

[3] Goldberg, A. and Rubin, K. *Succeeding with Objects: Decision Frameworks for Project Management*. Addison-Wesley Publishing Company, 1995.

[4] Kiczales, G. and Lamping, J. Issues in the Design and Specification of Class Libraries. In *Proceedings of OOPSLA '92* (Oct. 18-22, Vancouver, Britisch Columbia, Canada). ACM/SIGPLAN, New York, 1992, pp. 435-451.

[5] Opdyke, W.F. and Johnson, R.E. Creating Abstract Superclasses by Refactoring. In *Proceedings of CSC '93*, ACM Press, New York, 1993, pp. 66-72.

[6] Steyaert, P. and Lucas, C. and Mens, K. and D'Hondt, T. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA '96* (Oct. 6-10, San Jose, California). ACM/SIGPLAN, New York, 1996, pp. 268-285.