# Reuse Contracts: Making Systematic Reuse a Standard Practice

Patrick Steyaert, Carine Lucas, Kim Mens

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
Tel: (32) 2-629-3581,
Fax: (32) 2-629-3525
Email: prsteyae@vnet3.vub.ac.be, clucas@vnet3.vub.ac.be, kimmens@is1.vub.ac.be

**Abstract**

While object-orientation has had a large impact on the popularisation of reuse, reuse in OO is mostly ad hoc and thus lessons can be learned from the work on systematic reuse. On the other hand, the emphasis of object-orientation on iterative development can help in reducing the large up-front investments that are typically associated with systematic reuse. We claim that systematic reuse needs to be reconciled with iterative development in order to make reuse a standard practice. Such a reuse methodology must emphasise the co-operation between asset providers and asset reusers to control how assets *can be* reused, how assets *are* reused and how changes propagate from assets to applications during iterative development. We propose reuse contracts as the basis for such a methodology.

**Keywords:** Reuse contracts, change management, iterative development, object-orientation.

**Workshop Goals:** Cross-fertilisation between the OO community and the reuse community, get feedback on our ideas on reuse contracts from the reuse community.

**Working Groups:** Component Certification Tools, Frameworks and Processes; Reuse of Earliest Life-Cycle Artifacts.

# 1 Background

The Programming Technology Lab is involved in a number of research initiatives that aim to solve mostly technical problems that prevent the establishment of systematic reuse. Both our inspiration on which problems are most important to solve and the validation of our work come from joint projects with industrial partners. These projects are mostly aimed towards the development of frameworks for vertical markets (broadcasting planning, hypermedia, group decision support systems, ...). While, before, our work mainly focused on implementation aspects, now our goal is to develop a methodology for systematic reuse that covers all phases of the life-cycle and is formally based, as well as practically applicable.

# 2 Position

## 2.1 Issues in Making Reuse Standard Practice in OO

It is often claimed that artefacts created with object-oriented techniques are, by their very nature, more reusable than artefacts created with conventional means. Consequently object-orientation has had a large impact on the popularisation of reuse, and object-oriented methods hold many promises for making reuse a standard practice. However, because of the opportunistic - ad hoc - nature, the reuse potential obtained with object-oriented methods alone, leads to marginal gains only. The reuse community has emphasised that to obtain more substantial gains, a more systematic approach to reuse is necessary. Systematic reuse requires a shift from crafting one system at a time to the use of engineering principles for entire families of systems. However, object-oriented software engineering has been traditionally concerned with the development of custom built single systems. So, none of the currently used OOA/OOD methods [1], [2], [7], [14], [16] are suited for this kind of systematic software reuse.

It thus seems obvious that the solution lies in introducing the concept of systematic reuse in OOSE. One main disadvantage of systematic reuse, however, is that it requires a very large up-front investment and it seems to be in conflict with the iterative development process that is so typical for OO. We therefore suggest a methodology that combines the best of both worlds by introducing systematic reuse in the OOSE process, while controlling the necessity for large up-front investments by focusing on an iterative development process. The following criteria express how to reconcile the principles that underly systematic reuse with those that underly iterative development in OO.

- *Systematic reuse must be disciplined but not too coercive*

  Systematic reuse is based on building models that are reused in a disciplined manner. So, in contrast with what is customary today, "design reuse" does not simply mean taking an existing design, copying it and modifying it to some particular need, but rather the development of a design model of our software that can be reused in a *disciplined* fashion. Still, in current OOSE methodologies the issue of disciplined reuse of analysis and design as well as programming artefacts is almost entirely neglected.

  While reuse should not be ad hoc, it should *not* be *too coercive* either. A developer of reusable assets should provide a reuser with specifications that are powerful enough to enable reuse, while not overconstraining the reuser. For example, the typical black box reuse mechanisms that are so popular these days are considered too coercive, because they do not allow reusers to adapt their components before reuse.

- *Evolution is an inherent property of systematic reuse*

  Systematic reuse should not only recognise the need for a reusable asset to evolve both during its initial design and when it is being reused, it should actually advocate the development of a methodology for managing change in the process of engineering reusable software. The development of reusable assets is inherently an *evolutionary process*. It requires a number of iterations of building/modifying the asset and reusing it to see if it is properly reusable. To be able to leverage on the investment made in building an asset, reusers must be able to benefit from future improvements of the assets they reuse: proper evolution of reused assets should not invalidate previous reuse. In a similar vein, reuse should go beyond the act of copying out code fragments and adapting them to current requirements without regard for the evolution of the reused fragments. This implies the management of some kind of consistency in the evolution of reusable software, to prohibit different versions of a reusable asset from propagating through different applications. While systematic reuse should present an opportunity to reduce maintenance effort, a proliferation of versions actually increases it, as older versions of an asset behave differently than newer versions. The absence of change management mechanisms is recognised as an important inhibitor to successful reuse [5], [12], [16].

- *Notations for systematic reuse must be formal but understandable*

  Much of what is currently being proposed for object-oriented software reuse (frameworks, design patterns,...) lacks *formal* notation and rules. There is, for example, a general understanding that a framework may only be reused in a pre-defined way: the basic framework structures may not be violated. The rules to do so remain informal documentation however. When using only informal descriptions of reusable artefacts, reasoning about, for example, how to reuse an artefact, effort estimations, or the impact of changes to reusers can only be done by error prone informal reasoning and no discipline in the reuse of artefacts can be enforced. Disciplined reuse requires models expressed in a formal notation and formal rules on how to reuse an artefact based on such models.

  On the other hand, reusable artefacts are preferably reused by as many reusers as possible. This means that they need to be expressed in terms that are *understandable* by reusers. For example, the models that are used early in the life-cycle at the analysis phase must be expressed in terms that are understandable to end users. Moreover, different models are needed for the same artefact ranging from very abstract models for quick assessment of the feasibility of reuse to detailed models for the actual reuse. This suggests some form of layering.

  The art is in finding the right balance between descriptions that are easily understood and expressed, and descriptions that capture enough of the semantics of the asset being built and reused.

In the next subsection we propose reuse contracts as a methodology for systematic reuse that handles these issues.

## 2.2 Reuse Contracts

Based on the problems described above, it is our conjecture that substantial reuse requires a strong and well-defined co-operation between asset providers and asset reusers. We therefore suggest the introduction of explicit *reuse contracts* [15]. The asset provider must document that part of an asset that is relevant to reusers, while reusers must declare how an asset is actually reused. When reuse is based on such a formal contract, reusers can profit from the changes made to reusable assets because formal rules can be defined that facilitate change propagation. Providers can profit from the information given by reusers to make their assets more reusable. This not only solves the maintenance problem, caused by the proliferation of versions, but also assists in the iterative development, as crucial feedback from reusers is made available to the asset developers.

Because the best-known object-oriented reuse technique today is undoubtedly the use of abstract classes with inheritance, in [15] we focused on the problem of reuse of class hierarchies as a more tangible case to express the ideas behind reuse contracts. In that case, a reuse contract documents the design of a class by means of its specialisation interface [11]. This information is used by reusers to decide which methods have to be overridden and which methods can be inherited. Furthermore, reuse contracts can only be changed by means of formal *reuse operators*. These encode the different ways a class can be reused and adapted: extension, concretisation and refinement are design preserving operators and their inverses: cancellation, abstraction and coarsening are design breaching operators. Although not the only operators imaginable, they do coincide with the typical ways to reuse abstract classes. Together, reuse contracts and reuse operators record the protocol between the providers and users of a reusable asset (in this case, abstract classes). Simple formal rules were defined that signal possible conflicts in existing reusers (inheritors), when changes are made to the assets they are reusing (parent classes). Most of the possible conflicts can be directly expressed in terms of reuse contracts and 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 than previously possible. As reuse contracts can only be adapted by means of certain predefined reuse operators, reuse is disciplined. Moreover, since we also included design breaching operators, reuse is not too coercive.

Because conflicts upon change can be easily detected, reuse contracts help to predict the work effort in updating existing applications. Because they document what aspects possible reusers can rely on, they can also be used by developers of a reuse library to decide whether making a certain change to the reuse library is a good idea or not. In the same vein, conflict detection rules can be used to guide application developers both in understanding where testing is needed when the reusable asset has changed and how to fix the problems.

Up until now, reuse contracts were only fully developed for the reuse of classes in an inheritance hierarchy. Early results on the application of the same principles to reuse of multi-class components and state chart diagrams, however, show a lot of promise regarding the scalability of the approach. Similarly, while this first experiment with reuse contracts mainly concentrated on implementation conflicts, current work is being done to apply the same ideas to the earlier life-cycle phases. These preliminary results support the premise that reuse contracts have a lot of potential as a general reuse methodology.

# 3 Comparison

- *Frameworks, Design Patterns and Cookbooks*

  As was already mentioned in section 2.1 most state-of-the-art object-oriented reuse methodologies are insufficiently formally supported. This is e.g. the case for object-oriented frameworks [9], [3] that are usually documented through very informal documents. One way to additionally document frameworks is through design patterns [4], [8], [13], but these suffer the same lack of formal underpinnings. Cookbooks [10] that guide reusers step by step in the reuse process (the "customisation" process for frameworks) were suggested as another approach, but these suffer from the additional problem that they are overly coercive, i.e. they can only guide reuses that were specified up front. Reuse contracts provide a formally underpinned documentation, that still allows enough flexibility.

- *Facades and Variation Points*

  While, up until now, the issue of disciplined reuse was almost entirely neglected in object-oriented analysis and design techniques, recently new concepts as facades and variation points [6] were introduced to tackle this problem. These approaches try to discover possible points of evolution at the earliest phases of the life-cycle and incorporate them in the design. While a good starting point, we believe that a complete methodology should also incorporate the handling of unforeseen reuses and adaptations. That concern is one of the main contributions of reuse contracts to these other methodologies.

## References


[1] G. Booch. *Object-Oriented Analysis and Design with Applications, (Second Edition).* Benjamin/Cummings, Redwood City, CA, 1993.

[2] D. Coleman, P. Arnold, S. Bdoff, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-oriented Development: the Fusion method.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[3] S. Cotter and M. Potel. *Inside Talingent Technology.* Addison-Wesley, 1995.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addisson-Wesley, 1994.

[5] A. Goldberg and K. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management.* Addisson-Wesley, 1995.

[6] I. Jacobson, M.L. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addisson-Wesley, 1997.

[7] I. Jacobson, M.Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addisson-Wesley, 1992.

[8] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 63–76, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[9] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, "1(2), February 1988.

[10] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, pages 26–49, August/September 1988.

[11] John Lamping. Typing the specialization interface. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 201–214, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[12] C. Pancake. Object Roundtable, The Promise and the Cost of Object Technology: A Five-Year Forecast. *Communications of the ACM*, 38(10):32–49, October 1995.

[13] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addisson-Wesley, 1994.

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[15] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 268–285. ACM Press, 1996.

[16] E. Yourdon. *Object-Oriented System Design: An Integrated Approach.* Yourdon Press Computing Systems, Prentice Hall, 1994.

## 4 Biography

**Patrick Steyaert** holds a post-doc position at the Programming Technology Lab at the Vrije Universiteit Brussel, where he received his PhD in 1994 with work on object-oriented programming languages, reflection and semantics. His current interest is in the integration of reuse contracts in existing object-oriented modeling techniques and the development of a methodology for the interative development of reusable assets.

**Carine Lucas** is a teaching assistant and researcher at the Programming Technology Lab at the Vrije Universiteit Brussel. She received a Licentiate Degree in Computer Science from the Vrije Universiteit Brussel in 1991. While before she conducted research in design and typing of object-oriented languages, her PhD will focus on the use of reuse contracts for implementation, maintenance and refactoring.

**Kim Mens** is a teaching assistant and researcher at the Programming Technology Lab at the Vrije Universiteit Brussel. He received a Licentiate Degree in Mathematics from the Vrije Universiteit Brussel in 1992 and a Licentiate Degree in Computer Science from the same institution in 1994. He has previously been working on object-oriented programming calculi and on typing of object-oriented programming languages. His long-term research goal is to develop the formal foundations of a general reuse methodology, based on the concept of reuse contracts.