Chapter 8

Conclusion

8.1 Summary

In this dissertation reuse contracts were introduced as an extended interface description mechanism, allowing structured documentation of reusable components. Reuse contracts assist software engineers in understanding how a component can be reused and in managing the evolution of components. Reuse contracts are based on the observation that a more structured communication between reusable components and reusers is crucial to enable disciplined reuse. This is achieved by letting reusers document the assumptions they make about the components they reuse, which allows to check whether these assumptions are broken on evolution.

We introduced a basic reuse contract notation for describing reusable components that consist of several interacting participants. Eight basic reuse operators were given that allow modelling all possible changes to a reuse contract. These reuse operators enable expressing the assumptions different components make about each other. They thus help structure complex systems and form the basis for a mechanism for managing change propagation. A number of rules were set up to detect possible conflicts during change and to assist developers in assessing where and how to test and how to adjust applications. Because the basic operators were too fine grained to model real systems naturally, a structure was set up that allows operators to be combined into coarser-grained ones. A number of useful combinations were presented, but more importantly, it was demonstrated how combined operators can be constructed.

While the work in this dissertation focussed on a reuse contract model for software systems composed of co-operating participants, the working method followed here can be repeated for other systems. The following steps need to be taken:

1. Find an adequate interface representation and define well-formedness;

2. Distinguish a complete set of basic reuse operators. Provide modifier, applicability and result definitions for each operator and prove that the modifier preserves well-formedness;

3. Set up a model for change propagation by defining rules to detect different conflicts;

4. Elaborate by declaring combined operators and adding model features.

While the basic reuse contracts were described on a relatively abstract level, they were integrated in the UML to demonstrate their practical use. While the two versions of reuse contracts representing respectively class and object combinations allowed concrete experiments to be executed, their introduction also demonstrated how the basic reuse contract model can easily be extended to incorporate domain-specific features:

1. Map the concepts from reuse contracts to features in the application area. Investigate how this influences well-formedness and the reuse operators.

2. Add additional features from the application area to the model.

3. Adapt the existing model and the well-formedness and operator definitions to cope with the features added in step 2.

4. Introduce additional operators to cope with the features added in step 2 and optionally provide additional combined operators.

5. Investigate how the new operators of step 4 influence the conflicts and detect what possible new conflicts can arise.

Finally, through some experiments we demonstrated how reuse contracts can be used to assist in the software development process. Based on reuse contracts, tools can be constructed that support developers in tasks as diverse as customisation, evolution of components, quality assessment, program restructuring, ... One experiment demonstrated how reuse contract specifications can be instrumental in the iterative development of a reusable design.

8.2 Evaluation and Future Work

Reuse contracts as presented in this dissertation are a first, but important, step on the long road towards disciplined reuse. Let us evaluate the claims made in chapter one. There we declared:

- Reuse contracts can be used as structured documentation of reusable components and generally assist a software engineer in adapting components to particular needs.

- Reuse contracts encourage disciplined reuse without being too coercive and provide a vocabulary and notation to discuss reuse. They do this through simple models that are easy to learn and intuitive in use.

- On evolution, reuse contracts assist in assessing how much work is necessary to update previously built applications, where and how to test and how to adjust these applications.

The first and second claim were affirmed in chapter 6. We demonstrated there how reuse contracts help in managing evolving designs. One of the great benefits of reuse contracts is the simplicity of its models. This makes reuse contracts easy and intuitive to use by developers, which stimulates the deployment of disciplined reuse. It also makes the construction of tools rather straightforward. The drawback of this simplicity is that reuse contracts can only help us that much. As became clear in the ATS experiment as well as in other examples, the current model falls short in describing features as return values, typing, and so on. Section 8.2.1 discusses the usefulness and feasibility of some possible extensions.

The third claim above was also addressed by the ATS experiment in chapter 6 and by the tools. The ATS case demonstrated how reuse contracts can assist in evolution while chapter 7 discussed tools that can assist in assessing designs and keeping code and design consistent. Section 8.2.2 discusses some other possible uses.

Finally, besides the above elaborations, more work needs to be done on the model described in this dissertation on a theoretical as well as a practical level. This is discussed in section 8.2.3.

8.2.1 Possible Extensions

A large range of possible extensions to the current model are conceivable. They can be divided in two categories: extensions and adaptations of the current model and applications of the reuse contracts model to other areas that require a partially or completely new model.

Including Other Information

Private, Protected, Public A first straightforward extension of the current model is the addition of method annotations as the keywords private, protected and public. [Cor97] discusses the incorporation of reuse contracts in Java and the influence of these keywords on the change propagation model. The addition of those keywords is comparable to the addition of the keywords abstract and concrete in this dissertation. Extra operators can be introduced as well. What is interesting here is the influence on the conflicts. For example, declaring a method to be private prevents it from being captured.

[Cor97] also discussed the keywords final and static. Extensions as these are straightforward, because these are static properties to which compliance can easily be verified and of which the influence on inheritance and evolution are well-known.

Typing Information Another obvious extension is the incorporation of type information. In this dissertation methods were identified by a name, it is straightforward to extend names to method signatures. This was also done in the work on Java [Cor97].

Order, Branches and Loops In the modelling of object-oriented frameworks we noted that in describing object interactions, information on possible branches and loops or on the order in which methods are invoked can be useful. A notation could be conceived where specialisation clauses are not just sets of methods, but expressions including information on order and branches. Such a notation could be based on regular expressions or finite state machines. Compliance of the code to such expressions could still be checked statically if they are restricted to order, conditionals and loops. A bigger problem is defining the reuse operators on such expressions. When is one expression a refinement of the other? One rapidly runs into tractability problems when trying to answer such questions.

Result Values In the air time sales framework we also had problems to model the situation in which a method returned true on the superclass and false on one of the subclasses. In some specific cases checking this statically is possible[1], but generally we cannot include information on return values. Therefore we suggested introducing an extra operation *redefinition*, to express that something has changed in the code, but we cannot tell what. This conservative approach prevents us from giving more information about what kinds of conflicts might arise, but preserves us from needing data-flow analysis.

Dynamicity Similar problems might arise when further investigating dynamicity, for example, participant bindings changing at run-time or a dynamic number of participants. In chapter 5 we introduced implementation stereotypes in collaboration reuse contracts to denote the kind of binding. We mentioned there that extra well-formedness constraints could be introduced. For example, when we know that an acquaintance relationship has the stereotype ≪*parameter*≫, we know that this acquaintance relationship is only accessible from within the method of which it is an argument. UML also dinstinguishes dynamicity of participant bindings through implementation stereotypes ≪*new*≫, ≪*destroyed*≫ and ≪*transient*≫. More experiments are necessary to detect how much of this information can be checked. A lot will obviously depend on whether the language is statically typed or not.

[1] For example, in the air time sales framework, where the bodies literally showed ``return true'' and ``return false''.

Applying the Model to Other Areas

We expressed the basic reuse contract model in an abstract way, because we believe it can be applied to different kinds of component systems. However, further research is necessary in order to validate this claim. While we have applied the model to class and object combinations, we need to apply it to analysis models, architectural descriptions, component systems containing larger-grained components, ... Two interesting candidates are architecture description languages (ADLs) and interface definition languages (IDLs). Some adjustments might be needed to enable the application of our model to these languages.

Architecture Description Languages While in this dissertation we focused on components consisting of a number of classes, the same approach can be very useful on higher-level descriptions as provided by Architecture Description Languages (ADLs). Two different approaches are imaginable.

- In the first approach architecture descriptions are used as the extended interfaces in this dissertation and operators are defined on them. That way the relationship between different architectures can be expressed and the evolution of architectures can be monitored.
- Another approach could be to use a form of reuse contracts as connectors. This is related to other approaches that make interaction protocols between components explicit as separate entities next to the basic objects (such approaches were discussed in section 1.3.5). Using reuse contracts in such way is a rather different approach (e.g., should operators then be defined on reuse contracts alone or also on the components they connect ?) and requires more extensive research.

Interface Definition Languages As Interface Definition Languages (IDLs) provide language neutral and totally declarative descriptions of components in order to achieve language and platform-independent composition, they immediately arise as an ideal candidate on which to apply reuse contracts. Interface definitions specify the operations an object is prepared to perform, the input and output parameters they require, and exceptions that may be generated along the way. An extension of single class reuse contracts could therefore be made that incorporates, amongst others, exception handling and parameter specifications as `in`, `out` or `inout`. IDL also allows methods to be specified as `oneway` or not. This brings us to the subject of distribution.

Specific Areas In general, special flavours of reuse contracts can be conceived for different areas as distribution, error handling, ... For example, in a way similar to the addition of keywords specifying abstractness or visibility, keywords specifying synchronisation or locking could be added. Investigating how these different areas could be integrated in the basic model in an orthogonal way is a key area for future research.

State Diagrams In addition to the application of the model presented in this dissertation to related areas, we are convinced that the general principles of the reuse contract approach, i.e., a structural interface description with a complete set of basic reuse operators on top of which extensions can be built, is applicable to a much larger range of models. One experiment with state diagrams already proved to be promising [MS96]. There as well a notation for state diagrams was chosen, operators were defined on this notation and rules for change propagation were set up.

8.2.2 Other Uses

As became clear from the discussion in chapter 1, handling evolution of components is not the only inhibitor for systematic reuse. Other problems as how to handle complex object interactions and achieve separation of concerns, how to do effort estimations and how to measure reuse are equally important. This section discusses some domains where reuse contracts can also prove useful. The topics are ordered in increasing degree of complexity.

Ameliorating Design, Quality Assessment Currently reuse contracts are used to assist in modelling evolution of designs, but it is still up to the designer to decide what design choices are made. The information provided by reuse contracts could be used to decide on where the coupling between components is too high or the operators could assist in assessing the quality of template methods. If a template method is refined a number of times, possibly partially with the same invocations, this gives hints on how to ameliorate the design of the framework.

Metrics This is closely related to the topic of software metrics. The kinds of operators that appear often in customisations indicate how much the general design is respected or relaxed. Metrics could be set up based on this observation that indicate the reusability of a system.

Effort Estimation In the same vein, such metrics combined with the possibility to specify how a system can be customised might be useful to do effort estimations. One of the very hard problems in software engineering is estimating how long a project will take. This is crucial to the software business as underestimating means losses, while overestimating implies asking too high a price, possibly missing out on assignments. Classic measurement techniques based on the waterfall model fall short where object-oriented systems are concerned. A combination of metrics, the ability to assess impact analysis and the specification of possible customisations could prove helpful here as well.

8.2.3 Ameliorations to the Model

Besides elaborations on the work presented here, more work needs to be done on refining and tuning the current model.

The Reuse Contracts Model First work needs to be done on the theoretical level. A more rigourous, complete model of reuse contracts is crucial. The current notation, for example, is limited with respect to the detection of conflicts that involve the transitive closure of specialisation clauses. Furthermore, the trade-off between the ability to detect more conflicts and the flexibility of the reuse modifiers must be investigated further. Consider, for example, the suggestion to have two distinct operators for complete and partial participant concretisation instead of just the one. This trade-off is driven by the amount and the kind of information that is included in reuse modifiers. More research is necessary to establish a model where the kind of information that is included in modifiers is more generic, thus allowing the users to document their assumptions in a more flexible fashion. Similarly, when discussing the modelling of design patterns, we felt the need to have a more generic way of expressing reuse contracts.

Case Studies Second, more experimental research is needed, studying the role of reuse contracts in large, multi-person software developments. While small experiments are convincing with respect to the usefulness of the reuse contract approach, larger case studies are needed in order to test the scalability of the approach and to assess how reuse contracts can be extended and fine-tuned.

Since reuse contracts are designed to be used in different stages of the software process, different kinds of case studies are useful. A few possibilities are:

1. Documenting an existing framework with reuse contracts and expressing the customisation guidelines through reuse operators. It should then be measured how much this documentation aids customisers;

2. Documenting a system with reuse contracts and see how the rules assist software engineers when the system evolves;

3. Using extractors to document and evaluate existing systems and see how this can be used as a basis for re-engineering;

4. Experimenting with the quality assessment criteria and heuristics suggested in chapter 7.

Tools In order to perform larger case studies, tool support is absolutely crucial. The prototype of the extractor tool, the extension of the interface construct in Java and the restricted experiments on clustering and design assessment carried out on top of the extractor demonstrate how the reuse contract approach can lead to interesting results.

These experiments need to be extended to multi-class and collaboration contracts. Reuse contracts can be incorporated in an existing case tool. More interactive extractors must be developed and a repository must be created to store all extracted and otherwise provided information. On top of this a system can be built to assist in change propagation, based on the rules provided in this thesis. This system can automatically perform the kinds of verifications we performed manually in our experiment with the air time sales framework. Other elaborations, such as the design assessment tool, fit well into such an approach.

Methodology The long term goal of the combined research efforts described above is a full-fledged methodology for disciplined reuse, with a focus on evolution. To achieve this goal, software process studies are needed to better understand the role of reuse contracts in different phases. This is again related to carrying out case studies.

8.3 Main Contribution

The main contribution of this dissertation is its presentation of a general approach to disciplined reuse, based on well-documented communication between reusable components and their reusers. Reuse contracts provide a vocabulary and notation to discuss disciplined reuse, using simple models that are easy to learn and intuitive in use. Reuse contracts assist software engineers in understanding the structure and operational behaviour of software, in adapting components to particular needs and in estimating and managing the impact of changes on evolution.

Appendix A

Conflict Detection Rules

This appendix repeats the tables representing which conflicts can be caused by which operators. The tables were introduced in chapters 3 and 5.

	participant extension	context extension	participant ref./coars.	context ref./coars
participant extension	operation name	*no conflicts*	*no conflicts*	*no conflicts*
context extension	-	participant name	*no conflicts*	*no conflicts*
participant ref./coars	-	-	operation invocation	*no conflicts*
context ref./coars	-	-	-	acquaintance relationship

Table A.1: Interface Conflicts

	context cancellation	participant cancellation	context coarsening
any operator except context extension	dangling participant reference	-	-
part. cancellation part. refinement part. coarsening	-	dangling operation reference	-
part. extension part. refinement part. coarsening	-	-	dangling acquaintance reference

Table A.2: Dangling Reference Conflicts

	part. extension	part. refinement	part. coarsening
part. extension	accidental operation capture, unanticipated recursion	*no conflicts*	*no conflicts*
part. refinement	-	regular operation capture, unanticipated recursion	regular operation capture, inconsistent operations
part. coarsening	-	-	inconsistent operations

Table A.3: Conflicts concerning the Calling Structure

The conflicts that can be caused by extension, refinement, extending refinement and connected extension are the union of all conflicts that can be caused by their composing operators.

The set of conflicts that can be caused by factorisation is the union of all conflicts that can be caused by participant refinement, participant coarsening and participant extension, except for inconsistent operations.

	part. refinement	specialisation
part. extension	-	-
cont. extension	-	-
part. cancellation	dang. operation reference	dang. operation reference
cont. cancellation	dang. participant reference	dang. participant reference
part. refinement	*operation invocation*, unanticipated recursion, regular operation capture	unanticipated recursion, regular operation capture
cont. refinement	-	-
part. coarsening	*operation invocation*, inconsistent operations, regular operation capture	inconsistent operations, regular operation capture
cont. coarsening	dang. acquaintance reference	dang. acquaintance reference

Table A.4: Conflicts with Specialisation

The set of conflicts that can be caused by layered concretisation is the union of all conflicts that can be caused by participant refinement and participant concretisation.

	participant concretisation	participant abstraction
part. extension	incomplete implementation	-
cont. extension	-	-
part. cancellation	dang. operation reference	dang. operation reference
cont. cancellation	dang. participant reference	dang. participant reference
part. refinement	mixed method interface, regular method capture	mixed method interface, regular method capture
cont. refinement	-	-
part. coarsening	mixed method interface, inconsistent methods	mixed method interface, inconsistent methods
cont. coarsening	dang. acquaintance reference	dang. acquaintance reference
part. concretisation	annotation	incomplete implementation
part. abstraction	incomplete implementation	annotation

Table A.5: Conflicts with Participant Concretisation and Abstraction

Bibliography

[AWB+93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Object-based Distributed Processing, R. Guerraoui, O. Nierstrasz, M. Riveill (eds), LNCS 791*, pages 152–184. Springer-Verlag, 1993.

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications, (Second Edition)*. Benjamin/Cummings, Redwood City, CA, 1993.

[BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson. Unified Method Language 1.0. Technical report, Rational, 1997.

[CAB+94] 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.

[Cas94] E. Casais. Automatic reorganization of object-oriented hierarchies: a case study. *Object Oriented Systems*, 1:95–115, 1994.

[CHSV97] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. Evolving custom-made applications into domain-specific frameworks. *Communications of the ACM*, October 1997.

[Cor97] G. Cornelis. Reuse contracts as module system in statically typed object-oriented languages. Graduation report, Vrije Universiteit Brussel, Belgium, May 1997.

[CP95] S. Cotter and M. Potel. *Inside Taligent Technology*. Addison-Wesley, 1995.

[CS95] J. Coplien and D. Smith, editors. *Pattern Languages of Program Design*. Addison-Wesley Publishing Company, 1995.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, 2nd edition, 1991.

[DDN+97] S. Demeyer, S. Ducasse, R. Nebbe, O. Nierstrasz, and T. Richner. Using restructuring transformations to reengineer object-oriented systems, a position paper on the FAMOOS project. Technical report, Software Composition Group, University of Berne, Switzerland, 1997.

[Dem96] S. Demeyer. *Zypher, Tailorability as a Link from Object-Oriented Software Engineering to Open Hypermedia.* PhD thesis, Vrije Universiteit Brussel, Departement Informatica, 1996.

[DR96] R. Johnson B. Opdyke D. Roberts, J. Brant. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[DR97] S. Ducasse and T. Richner. Connectors as First-Class Objects in FLO, FAMOOS project position paper. Technical report, Software Composition Group, University of Berne, Switzerland, 1997.

[FCDR95] I.R. Forman, M.H. Conner, S.H. Danforth, and L.K. Raper. Release-to-Release Binary Compatibility in SOM. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, pages 426–438. ACM Press, 1995.

[Fra94] W. Frakes. Systematic software reuse: a paradigm shift. In *Third International Conference on Software Reuse, Advances in Software Reusability*, pages 2–3. IEEE Computer Society Press, 1994.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addisson-Wesley, 1994.

[Gla95] M. Glandrup. Extending C++ using the concepts of composition filters. Master's thesis, University of Twente, November 1995.

[GR95] A. Goldberg and K. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management.* Addisson-Wesley, 1995.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering, V. Ambriola and G. Tortora (eds.)*, volume I. World Scientific Publishing, 1993.

[GS96] D. Garlan and M. Shaw. *Software Architecture: Perspectives on an Emerging Discipline.* PrenticeHall, 1996.

[Hal96] A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66–76, March 1996.

[HHG90] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *ECOOP/OOPSLA'90 Proceedings.* ACM Press, 1990.

[Hol92] I. Holland. *The Design and Representation of Object-Oriented Components.* PhD thesis, Northeastern University, 1992.

[HS95] W.L. Hürsch and L.M. Seiter. Automating the evolution of object-oriented systems. Technical Report NU-CCS-95-06, College of Computer Science, Northeastern University, Boston, MA, April 1995.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addisson-Wesley, 1992.

[JF88] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), February 1988.

[JGJ97] I. Jacobson, M.L. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addisson-Wesley, 1997.

[Joh92] R.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.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., 2nd edition, 1990.

[KC94a] P. Kogut and P. Clements. Features of architecture representation languages. Technical Report CMU/SEI-94-TR-tbd, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1994.

[KC94b] P. Kogut and P. Clements. The software architecture renaissance. *STSC's Crosstalk Newsletter*, October 1994.

[KdRB91] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KKS96] N. Klarlund, J. Koistinen, and M. Schwartzbach. Formal design constraints. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 370–383, 1996.

[KL92] G. Kiczales and J. Lamping. Issues in the design and documentation of class libraries. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 435–451, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[Koe95] A. Koenig. Patterns and antipatterns. *Journal of Object-Oriented Programming*, March 1995.

[KP88] 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*, 1(3), 1988.

[KY95] N. Krol and D. Yockelson. CI Labs OpenDoc White Paper. Technical report, CILabs, 1995.

[Lal89] W.R. Lalonde. Designing families of data types using exemplars. *Transactions on Programming Languages and Systems*, 11(2), 1989.

[Lam93] J. Lamping. Typing the specialization interface. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 201–214, oct 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[Lea95] G. T. Leavens. Larch/C++ reference manual. version 5.1. Technical Report Available in ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz, Iowa State University, 1995.

[LH89] K. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[Lie95] K. Lieberherr. *Adaptive Object-Oriented Software, the Demeter Method with Propagation Patterns*. PWS Publishing Company, 1995.

[Lie96] K. Lieberherr. Private communication, 1996.

[Lin94] R.C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–58, March 1994.

[LRD+88] K. Lee, M. Rissman, R. D'Ippolito, C. Plinta, and R. Van Scoy. An OOD Paradigm for flight simulators. Technical Report CMU/SEI-88-TR-30, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1988.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science, C.A.R. Hoare, Series Editor. Prentice Hall, 1988.

[Mez97] M. Mezini. Maintaining the consistency of class libraries during their evolution. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, 1997.

[Moo96] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 235–250, 1996.

[MS96] T. Mens and P. Steyaert. Incremental design of layered state diagrams. Technical Report vub-prog-tr-97-04.ps.Z, Vrije Universiteit Brussel, Belgium, 1996.

[MSW97] T. Murer, D. Sherer, and A. Wurtz. Improving component interoperability. In *Special Issues in Object-Oriented Programming, Workshop Reader on the 10th European Conference on Object-Oriented Programming (ECOOP '96), ed. Max Muhlhauser*, pages 150–158. dpunkt Verlag, 1997.

[Nei94] J.M. Neighbors. An assessment of reuse technology after ten years. In *Third International Conference on Software Reuse, Advances in Software Reusability*, pages 6–13. IEEE Computer Society Press, 1994.

[NT95] O. Nierstrasz and D. Tsichritzis. *Object-Oriented Software Composition*. Prentice Hall, 1995.

[OB97] A. Olafsson and D. Bryan. On the need for "required interfaces" of components. In *Special Issues in Object-Oriented Programming, Workshop Reader on the 10th European Conference on Object-Oriented Programming (ECOOP '96), ed. Max Muhlhauser*, pages 159–165. dpunkt Verlag, 1997.

[Obj91] *ObjectWindows for C++ User's Guide*, 1991.

[OH97] R. Orfali and D. Harkey. *Client/Server Programming with Java and Corba*. John-Wiley and Sons, Inc., 1997.

[Opd92] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pan95] 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.

[PDe91] R. Prieto-Diaz and G. Arango (eds.). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.

[Pin95] X. Pintado. Gluons and the cooperation between software components. In *Object-Oriented Software Composition, O. Nierstrasz and D. Tsichritzis (eds.)*, 1995.

[Pre94] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addisson-Wesley, 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RG92] K. Rubin and A. Goldberg. Object behaviour analysis. *Communications of the ACM, Special Issue on Object-Oriented Methodologies*, September 1992.

[Sie96] J. Siegel. *CORBA Fundamentals and Programming.* John Wiley and Sons, Inc., 1996.

[Spi89] J. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, January 1989.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.

[Sun97] Sun Microsystems, JavaSoft. *JavaBeans 1.0 API Specification*, 1997.

[SW96] R.J. Stroud and Z. Wu. Using metaobject protocols to satisfy non-functional requirements. In *Advances in Object-Oriented Metalevel Architectures and Reflection, C. Zimmerman (eds.)*, 1996.

[Tur91] *Turbo Vision for C++ User's Guide, Tutorials, Class Library reference*, 1991.

[VCK96] J. Vlissides, J. Coplien, and J. Kerth, editors. *Pattern Languages of Program Design 2.* Addison-Wesley Publishing Company, 1996.

[vDM95] W. van Dijck and J. Mordhorst. Composition filters in Smalltalk. Graduation report, HIO Enschede, The Netherlands, May 1995.

[VV96] A. Vercammen and W. Verachtert. Psi: From custom developed application to domain specific framework. In *In Addendum to the proceedings of OOPSLA '96*, 1996.

[WBW89] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: A responsibility driven approach. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 71–75, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices.

[WGM89] A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++ - a seamless object-oriented application framework. *Structured Programming*, 10(2), 1989.

[Wil96] J.D. Williams. Managing iteration in OO projects. *IEEE Software*, pages 39–43, September 1996.

[You94] E. Yourdon. *Object-Oriented System Design: An Integrated Approach.* Yourdon Press Computing Systems, Prentice Hall, 1994.