This dissertation shows how role modeling can be integrated with traditional class-based modeling so that the respective strengths are added and the weaknesses are dropped.

  - The dissertation revises the existing concepts of role, role type, role model, class, and class model to better fit together, and is precise about the distinction between type level (role type, class, role model, class model) and instance level (role, object collaboration task, object collaboration).
  - The dissertation demonstrates the complementary focus of classes and role models. A role model describes how objects collaborate for one specific object collaboration task. A class defines how several roles from different collaboration tasks come together in one object, thereby defining how to bridge and integrate the tasks.

- *Introduction of an explicit framework concept.* The modeling method gives a precise definition of what a framework is and what its properties are. The definition of the framework concept is based on the notions of free role model, built-on class set, and extension point classes.
  - Using free role models, developers specify how a framework is to be used by clients.
  - Using built-on class sets and free role models, developers specify how a framework builds on other frameworks and how it depends on its environment. While software engineering has long understood that next to provided functionality also required functionality needs to be specified [Wir82, PN86], facilities to do so in framework design have been missing.
  - Using extension-point class sets, developers can specify how a framework may be extended.

  These concepts are unique to frameworks and let developers speak about and deal with frameworks in a framework-specific way. A framework becomes an explicit design artifact rather than just another class model.

In addition, role modeling for framework design provides excellent means for describing design patterns and for showing how they are used in the context of object-oriented frameworks.

- *Description of design patterns.* Role modeling as defined in this dissertation is a more general way of illustrating design patterns. A role model illustration of a pattern can be applied in more ways than a class-based illustration. A role model illustration adds to a class-based illustration, because a class-based illustration typically suggests too rigid a structure of pattern application.
- *Application of patterns in framework design.* Because role models are prepared for composition right from the start, they show well how pattern applications compose and overlap in framework design. Class-based modeling offers no such facilities. Annotating classes as participants of a pattern instantiation goes into the right direction but stops halfway. Role modeling goes all the way.

These contributions are described in more detail and validated in the main body of the dissertation. Some of them have also been published at conferences and in journals [Rie96a, Rie97c, RG98, RBGM99].

10.2 Future work

This dissertation work opens several venues for future work, both with a narrow focus on role modeling, and a larger focus on frameworks and design patterns.

- *Choice of a type specification mechanism.* The dissertation suggests no specific type specification mechanism. Any mechanism that provides types, subtyping, and type composition suffices. How-

ever, some type specification mechanisms may be more convenient and more effective to use than others. Therefore, a mechanism custom-tailored to the needs of role modeling may be developed.

- *Introducing dynamic behavior specifications.* Because type specification issues are largely ignored in this dissertation, not much is said about dynamic behavior specifications. However, role models are best described not only by individual role types, but also by descriptions of the allowed collaborative behavior of objects acting according to the role types. Therefore, Reenskaug's or Andersen's mechanism to describe collaborative behavior of object roles may be adapted [Ree96, And97], or a new one may be developed.

- *Composition of role types independently of classes.* Role modeling for framework design as presented in this dissertation composes role types to derive classes. I have never found a real need to compose role types to become composite role types. However, it may be a nice-to-have feature that could be useful once it is available.

Future work of this kind may directly build on current type specification mechanisms. However, existing mechanisms need to be adapted to fit role modeling in such a way that the separation of concerns achieved by role modeling is maintained. The benefit of reduction in complexity that role modeling achieves is directly based on the separation of concerns it provides.

- *Inheritance relationship between role models (addressing the problem of covariant redefinition).* The dissertation does not introduce a concept of inheritance between role models. Any role model that could be viewed as a specialization of an existing role model is viewed as a different unrelated role model. It seems helpful in many situations, however, to view one role model as a specialization of another more general role model. For example a simple PersonModel/PersonView role model might be specialized to form a CustomerModel/CustomerView role model.

  Inheritance between role models might give a new twist to the problem of covariant redefinition of operation signatures. The covariant redefinition of parameters of an operation in a subclass (and the contravariant redefinition of return value or object types) serves to ensure that class hierarchies are specialized in parallel. The covariant redefinition of operations of a class is always carried out with a particular partner class in mind. Thus, covariant redefinition is about ensuring constraints on a set of allowed collaboration tasks (rather than individual classes). Expressing these collaboration tasks is all what role models are about.

- *Extension of programming languages with role modeling concepts.* It would certainly be helpful to support the implementation of a role–model–based design with dedicated programming constructs. Such a role-oriented programming language might provide concepts for directly and conveniently expressing role types and role models.

  Current work already points into that direction. Van Hilst presents a role programming method using C++ templates [Van97], and Kendall uses aspect-oriented programming to more easily implement role–model–based designs [Ken99].

- *Support for design patterns and design templates.* Role modeling lets developers more easily apply and recognize design patterns in object-oriented designs (than is possible with traditional class-based designs). A dedicated design notation for specifying design templates for design patterns may be based on role modeling rather than class-based modeling [Rie96a]. Then, the application of a design pattern leads to a specific role model that can be easily composed with other role models in the context of an object-oriented design.

  On the pattern/template level, something alike to composite role types is going to be helpful, as illustrated by the Bureaucracy pattern [Rie98]. The Bureaucracy pattern is a composite pattern in which different role types from different design patterns are composed to form the pattern/template level equivalent of a composite role type.

- *Empirical assessment of use of design patterns in framework design.* The case studies of Chapters 6 to 8 provide some empirical data about the frequency of use of design patterns in framework de-

sign. Role models can be used as a kind of object-oriented function-point, that is, as an atomic unit of functionality in framework design. Thus, role models may serve as a coarse-grained measure for complexity in framework design.

An analysis of frameworks described using role modeling can provide us with a figure about the frequency of design pattern application in relation to the overall functionality (= total number of role models in a framework's design). Arriving at a statement like "60% of a framework's functionality can be described using design patterns" is a valuable result to justify further research into design patterns.

Finally, marrying component-based design with object-oriented frameworks is a whole new research area. It does not follow directly from this dissertation, but it should take the new understanding of frameworks gained through this dissertation into account. Then, role modeling is likely to find its way into component design and implementation. Role modeling might even be reintroduced on a component framework level to better describe how components collaborate.

10.3 Final conclusions

Role modeling provides separation of concerns in a way that is highly beneficial to class-based design of frameworks. This dissertation shows how to marry role modeling with class-based design of frameworks. Role modeling for framework design has the following properties:

- It reduces complexity of classes in framework design.
- It reduces complexity of object collaboration in framework design.
- It better supports specifying requirements put upon use-clients of a framework.
- It lets developers apply and recognize design patterns in frameworks more easily.
- It provides new and revised role modeling concepts for more precise framework design.
- It makes frameworks first class citizens of software architecture.

Role modeling for framework design combines the strengths of role modeling with those of class-based modeling while leaving out their weaknesses. It is therefore an evolutionary extension of current methods that preserves existing investments. Finally, role modeling for framework design is the first comprehensive modeling method to make frameworks explicit design artifacts.

A

References

ABGO93 A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. “An Object Data Model with Roles.” In *Proceedings of the 19th International Conference on Very Large Databases.* San Mateo, CA: Morgan Kaufmann, 1993. Page 39-51.

AC96 Martin Abadi and Luca Cardelli. “On Subtyping and Matching.” *ACM Transactions on Programming Languages and Systems* 18, 4 (July 1996). Page 401-423.

AG96 Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

All97 Robert J. Allen. *A Formal Approach to Software Architecture.* Ph.D. Thesis, CMU-CS-97-144. Pittsburgh, PA: Carnegie Mellon University, 1997.

And97 Egil P. Andersen. *Conceptual Modeling of Objects.* Ph.D. Thesis. Oslo, Norway: University of Oslo, 1997.

Bäu98 Dirk Bäumer. *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme.* Dissertation. Hamburg, Germany: Universität Hamburg, 1998.

BBE95 Andreas Birrer, Walter Bischofberger, and Thomas Eggenschwiler. “Wiederverwendung durch Framework-Technik-Vom Mythos zur Realität”. *OBJEKTSpektrum* 5 (1995). Page 18-26.

BC98 Kent Beck and Ward Cunningham. “A Laboratory for Teaching Object-Oriented Thinking.” In *Proceedings of the 1989 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA ’89). ACM Press, 1989. Page 1-6.

BCC+96 Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch, and John Vlissides. “Industrial Experience with Design Patterns.” In *Proceedings of the 18th International Conference on Software Engineering* (ICSE 18). IEEE Press, 1996. Page 103-114.

BCG95 William Berg, Marshall Cline, and Mike Girou. “Lessons Learned from the OS/400 OO Project.” *Communications of the ACM* 38, 10 (October 1995): 54-64.

Be97 Be, Inc. *Be Developer Guide.* O’ Reilly, 1997.

BGK+97 Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, and Heinz Züllighoven. “Framework Development for Large Systems.” *Communications of the ACM* 40, 10 (October 1997). Page 52-59.

BGR96a Walter Bischofberger, Michael Guttman and Dirk Riehle. “Architecture Support for Global Business Objects: Requirements and Solutions.” In *Joint Proceedings of the SIGSOFT ’96 Workshops* (ISAW-2). Edited by Laura Vidal, Anthony Finkelstein, George Spanoudakis, and Alexander L. Wolf. ACM Press, 1996. Page 143-146.

BGR96b Walter Bischofberger, Michael Guttman and Dirk Riehle. “Global Business Objects: Requirements and Solutions.” In *Proceedings of the Ubilab Conference '96, Zürich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Germany: Universitätsverlag, 1996. Page 79-98.

BHH+97 Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. “Towards a Formalization of the Unified Modeling Language.” In *Proceedings of the 1997 European Conference on Object-Oriented Programming* (ECOOP ’97). Springer Verlag, 1997. Page 344-366.

BHKS97 Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. “A Graphical Description Technique for Communication in Software Architectures.” In *Software Architectures and Design Patterns in Business Applications.* Edited by Manfred Broy, Ernst Denert, Klaus Renzel, and Monika Schmidt. Technical Report TUM-I9746. Munich, Germany: Technische Universität München, 1997.

BMR+96 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

Box98 Don Box. *Essential COM.* Addison-Wesley, 1998.

BR98 Dirk Bäumer and Dirk Riehle. “Product Trader.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 29-46.

BRSW00 Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. “Role Object.” In *Pattern Languages of Program Design 4.* Edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison-Wesley, 2000. Page 15-32. Originally published as: Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. “Role Object.” In *Proceedings of the 1997 Conference on Pattern Languages of Programming* (PLoP ’97). Washington University Department of Computer Science, Technical Report WUCS-97-34, 1997. Paper 2.1.

BRS+98 Dirk Bäumer, Dirk Riehle, Wolf Siberski, Carola Lilienthal, Daniel Megert, Karl-Heinz Sylla, and Heinz Züllighoven. *Values in Object Systems.* Ubilab Technical Report 98.10.1. Zurich, Switzerland: UBS AG, 1998.

CIM92 Roy H. Campbell, Nayeem Islam, and Peter Madany. “Choices, Frameworks and Refinement.” *Computing Systems* 5, 3 (Summer 1992): 217-257.

Cox87 Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1987.

CP95 Sean Cotter, with Mike Potel. *Inside Taligent Technology.* Addison-Wesley, 1995.

DH72 Ole-Johan Dahl and C. A. R. Hoare. “Hierarchical Program Structures.” In *Structured Programming*. Edited by Ole-Johan Dahl, Edsger W. Dijkstra and C. A. R. Hoare. Academic Press, 1972.

DW98 Desmond F. D' Souza and Alan C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley Longman, 1998.

EF97 Eric Evans and Martin Fowler. "Specifications." In *Proceedings of the 1997 Conference on Pattern Languages of Programming* (PLoP '97). Washington University Department of Computer Science, Technical Report WUCS-97-34, 1997. Paper 2.3.

FHG98 Donald Firesmith, Brian Henderson-Sellers, and Ian Graham. *OPEN Modeling Language.* Cambridge University Press, 1998.

FK97 Robert G. Fichman and Chris F. Kemerer. "Object Technology and Reuse: Lessons from Early Adopters." *Computer* 30, 10 (October 1997). Page 47-59.

FS97 Mohamed E. Fayad and Douglas C. Schmidt (editors). Special Issue on Object-Oriented Application Frameworks. *Communications of the ACM* 40, 10 (October 1997).

FSJ99 Mohamed Fayad, Douglas Schmidt, and Ralph Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* Wiley & Sons, 1999.

Gam98 Erich Gamma. "Advanced Design with Patterns and Java." Tutorial given at the *1998 European Conference on Java and Object Orientation.* Copenhagen, Denmark, 1998. See Appendix E for a pointer to the tutorial.

GHJV95 Erich Gamma. Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

GR89 Adele Goldberg and David Robson. *Smalltalk-80—The Language.* Addison-Wesley, 1989.

GSR96 Georg Gottlob, Michael Schrefl, and Brigitte Röck. "Extending Object-Oriented Systems with Roles." *ACM Transactions on Information Systems* 14, 3 (July 1996). Page 268-296.

Hal96 Terry Halpin. "Business Rules and Object Role Modeling." *Database Programming and Design* 9, 10. San Mateo, CA: Miller Freeman. Page 66-72.

Hal98 Terry Halpin. "UML Data Models from an ORM Perspective." *Journal of Conceptual Modeling* (http://www.inconcept.com/jcm), April 1998.

HHG90 Richard Helm, Ian M. Holland and Dipayan Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." In *Proceedings of the 1990 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '90). ACM Press, 1990. Page 169-180.

HO93 William Harrison and Harold Ossher. "Subject-Oriented Programming (A Critique of Pure Objects)." In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '93). ACM-Press, 1993. Page 411-428.

HS88 Daniel Hoffman and Richard T. Snodgrass, "Trace Specifications: Methodology and Models" *IEEE Transactions on Software Engineering* 14, 9 (September 1988). Page 1243-1252.

Hür94 Walter L. Hürsch. "Should Superclasses be Abstract?" In *Proceedings of the 1994 European Conference on Object-Oriented Programming* (ECOOP '94, LNCS 821). Edited by Mario Tokoro and Remo Pareschi. Springer-Verlag, 1994. Page 12-31.

JF88 Ralph E. Johnson and Brian Foote. "Designing Reusable Classes." *Journal of Object-Oriented Programming* 1, 2 (June/July 1988). Page 22-35.

Joh92 Ralph E. Johnson. "Documenting Frameworks using Patterns." In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '92). ACM Press, 1992. Page 63-70.

JW98 Ralph Johnson and Bobby Woolf. “Type Object.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 47-65.

Ken99 Elisabeth A. Kendall. “Role Model Designs and Implementations with Aspect Oriented Programming.” Unpublished manuscript.

KLM+97 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-Oriented Programming.” In *Proceedings of the 1997 European Conference on Object-Oriented Programming* (ECOOP ’97). Springer Verlag, 1997. Page 220-242.

KO96a Bent Bruun Kristensen and Kasper Osterbye. “Roles: Conceptual Abstraction Theory and Practical Language Issues.” *Theory and Practice of Object Systems* 2, 3 (1996). Page 143-160.

Lew95 Ted Lewis (editor). *Object-Oriented Application Frameworks.* Greenwich: Manning, 1995.

LH89 Karl J. Lieberherr and Ian M. Holland. “Assuring Good Style for Object-Oriented Programs.” *IEEE Software* 22, 9 (September 1989). Page 38-48.

Lie95 Karl J. Lieberherr. *Adaptive Object-Oriented Software.* Boston, MA: PWS Publishing Company, 1995.

LKA+95 David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. “Specification and Analysis of System Architecture Using Rapide.” *IEEE Transactions on Software Engineering* 21, 4 (April 1995). Page 336-355.

Lop97 Cristina Lopes. *D: A Language Framework for Distributed Programming.* Ph.D. Thesis. Boston, MA: Northeastern University, 1997.

LW93a Barbara Liskov and Jeannette Wing. “Specifications and Their Use in Defining Subtypes.” In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’93). ACM Press, 1993. Page 16-28.

LW93b Barbara Liskov and Jeannette Wing. “A New Definition of the Subtype Relation.” In *Proceedings of the 1993 European Conference on Object-Oriented Programming* (ECOOP ’93). LNCS-707. Springer-Verlag, 1993. Page 118-141.

LW94 Barbara H. Liskov and Jeannette M. Wing. “A Behavioral Notion of Subtyping.” *ACM Transactions on Programming Languages and Systems* 16, 6 (November 1994). Page 1811-1841.

Mac82 B. J. MacLennan. “Values and Objects in Programming Languages.” *ACM SIGPLAN Notices* 17, 12 (December 1982). Page 70-79.

McA95a Jeff McAffer. *A Meta-Level Architecture for Prototyping Object Systems.* Ph.D. Thesis. Tokyo, Japan: University of Tokyo, 1995.

McA95b Jeff McAffer. “Meta-level Programming with CodA.” In *Proceedings of the 1995 European Conference on Object-Oriented Programming* (ECOOP ’95). LNCS-952. Springer-Verlag, 1995. Page 190-214.

Mey91 Bertrand Meyer. “Design by Contract.” *Advances in Object-Oriented Software Engineering*. Edited by Dino Mandrioli und Bertrand Meyer. Prentice-Hall, 1991. Page 1-50.

Mey92 Bertrand Meyer. *Eiffel. The Language*. Prentice-Hall, 1992.

Mey92b Bertrand Meyer. “Applying Design By Contract.” *IEEE Computer* 25, 10 (October 1992). Page 40-51.

MDEK95 Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. “Specifying Distributed Software Architectures.” In *Proceedings of 5th European Software Engineering Conference* (ESEC ’95). Springer-Verlag, 1995.

Ode98 James J. Odell. *Advanced Object-Oriented Analysis and Design Using UML*. Cambridge University Press, 1998.

OKH+95 Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. “Subject-Oriented Composition Rules.” In *Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’95). ACM Press, 1995. Page 235-250.

PN86 Ruben Prieto-Diaz and James M. Neighbors. "Module Interconnection Languages." Journal of Systems and Software 6, 4 (November 1986): 307-334.

RAB+92 Trygve Reenskaug, Egil P. Andersen, Arne Jorgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Nêss-Ulseth, Gro Oftedal, Anne Lise Skaar, and Pâl Stenslet. “OORASS: seamless support for the creation and maintenance of object-oriented systems.” *Journal of Object-Oriented Programming* 5, 6 (October 1992). Page 27-41.

RBGM99 Dirk Riehle, Roger Brudermann, Thomas Gross, and Kai-Uwe Mätzel. “Pattern Density and Role Modeling of an Object Transport Service.” *ACM Computing Surveys* 31, 2 (June 1999). To appear.

RD99a Dirk Riehle and Erica Dubach. “Working with Java Interfaces and Classes. Part 1.” *Java Report* 4, 7 (July 1999). Page 35pp.

RD99b Dirk Riehle and Erica Dubach. “Working with Java Interfaces and Classes. Part 2.” *Java Report* 4, 10 (October 1999). Page 34pp

Ree96 Trygve Reenskaug, with Per Wold and Odd Arild Lehne. *Working with Objects*. Greenwich: Manning, 1996.

Rie96a Dirk Riehle. “Describing and Composing Patterns Using Role Diagrams.” In *Proceedings of the 1996 Ubilab Conference, Zürich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Germany: Universitätsverlag Konstanz, 1996. Page 137-152. Originally published in *Proceedings of the 1st International Conference on Object-Orientation in Russia* (WOON ’96). Edited by Alexander V. Smolyaninov and Alexei S. Shestialtynov. St. Petersburg, Russia: Electrotechnical University, 1996. Page 169-178. See Appendix E for a pointer to this publication.

Rie96c Dirk Riehle. “Patterns for Encapsulating Class Trees.” In *Pattern Languages of Program Design 2.* Edited by John M. Vlissides, James O. Coplien and Norman L. Kerth. Addison-Wesley, 1996. Page 87-104.

Rie97a Dirk Riehle. *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose.* Ubilab Technical Report 97.1.1. Zürich, Switzerland: Union Bank of Switzerland, 1997.

Rie97c Dirk Riehle. “Composite Design Patterns.” In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’97). ACM Press, 1997. Page 218-228.

Rie97d Dirk Riehle. “Arbeiten mit Java-Schnittstellen und –Klassen (Teil 1 von 2).” *Java Spektrum* 5/97 (September/October 1997). Seite 26-33.

Rie97e Dirk Riehle. “Arbeiten mit Java-Schnittstellen und –Klassen (Teil 2 von 2).”. *Java Spektrum* 6/97 (November/Dezember 1997). Seite 35-43.

Rie98 Dirk Riehle. “Bureaucracy.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 163-186.

Rit97 Antonio Rito Silva. “Framework, Design Patterns, and Pattern Language for Object Concurrency.” In *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA ’97).

RS95 Dirk Riehle and Martin Schnyder. *Design and Implementation of a Smalltalk Framework for the Tools and Materials Metaphor.* Ubilab Technical Report 95.7.1. Zürich, Switzerland: Union Bank of Switzerland, 1995.

RSB+98 Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. “Serializer.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 293-312.

RZ95 Dirk Riehle and Heinz Züllighoven. “A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor.” In *Pattern Languages of Program Design.* Edited by James O. Coplien and Douglas C. Schmidt. Addison-Wesley, 1995. Page 9-42.

RZ96 Dirk Riehle and Heinz Züllighoven. “Understanding and Using Patterns in Software Development.” *Theory and Practice of Object Systems* 2, 1 (1996). Page 3-13.

Sch98 Bruno Schäffer. *Design and Implementation of Smalltalk Mixin Classes.* Ubilab Technical Report 98.11.1. Zurich, Switzerland: UBS AG, 1998.

SBF96 Steve Sparks, Kevin Benner, and Chris Faris. “Managing Object-Oriented Framework Reuse.” *Computer* 29, 9 (September 1996): 52-61.

SG96 Mary Shaw and David Garlan. *Software Architecture—Perspectives on an Emerging Discipline.* New Jersey: Prentice Hall, 1996.

Sie96 Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley & Sons, 1996.

Som98 Peter Sommerlad. “Manager.” In *Pattern Languages of Program Design 3.* Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 19-28.

Str94 Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

Sun96a James Gosling, Frank Yellin, and the Java Team. *The Java Application Programming Interface, Volume 1.* Addison-Wesley, 1996.

Sun96b James Gosling, Frank Yellin, and the Java Team. *The Java Application Programming Interface, Volume 2.* Addison-Wesley, 1996.

Szy98 Clemens Szyperski. *Component Software-Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

Tal95 Taligent Inc. *The Power of Frameworks.* Addison-Wesley, 1995.

UML97a Rational Software Corporation et al. *UML v1.1 Semantics.* Santa Clara, CA: Rational Software Corporation, 1997.

UML97b Rational Software Corporation. *UML v1.1 Notation Guide.* Santa Clara, CA: Rational Software Corporation, 1997.

Van97 Michael VanHilst. *Role Oriented Programming for Software Evolution.* Ph.D. Thesis. Seattle, WA: University of Washington, 1997.

Vli98 John Vlissides. *Pattern Hatching.* Addison-Wesley Longman, 1998.

VN96 Michael VanHilst and David Notkin. “Using Role Components to Implement Collaboration-Based Designs.” In *Proceedings of the 1996 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA ’96). ACM Press, 1996. Page 359-369.

WG95 André Weinand and Erich Gamma. “ET++ A Portable Homogeneous Class Library and Application Framework.” In *Object-Oriented Application Frameworks.* Edited by Ted Lewis. Greenwich: Manning, 1995. Page 154-194.

WGM89 André Weinand, Erich Gamma, and Rudolf Marty. “Design and Implementation of ET++, a Seamless Object-Oriented Application Framework.” *Structured Programming* 10, 2 (Juni 1989): Page 63-87.

Wir82 Wirth, N. (1982). Programming in Modula-2. Berlin, Heidelberg: Springer-Verlag.

WJS95 Roel Wieringa, Wiebren de Jonge, and Paul Spruit. “Using Dynamic Classes and Role Classes to Model Object Migration.” *Theory and Practice of Object Systems* 1, 1 (1995) Page 61-83.

Woo98 Bobby Woolf. “Null Object.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 5-18.

WSP+92 Peter Wegner, William Scherlis, James Purtilo, David Luckham and Ralph Johnson. “Object-Oriented Megaprogramming.” In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’92). ACM Press, 1992. Page 392-396.

WWW90 Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

WZ88 Peter Wegner and Stanley B. Zdonik. “Inheritance as an Incremental Modification Mechanism or What Like Is and Isn’t Like.” In *Proceedings of the 1988 European Conference on Object-Oriented Programming* (ECOOP ’88). LNCS 322. Springer-Verlag, 1998. Page 55-77.

B

Glossary

Class

A class is the definition of a (possibly infinite) set of objects, called its instances. A class defines a non-empty set of role types, a composition function, and a class type. The composition function, applied to all role types, results in the class type.

See Definition 3-16 and 3-2.

Class, built-on

A built-on class of a framework is the class of a built-on object. It is connected to the framework through one or more role models.

See Definition 4-8.

Class, extension

An extension class of a framework is a subclass of an extension-point class of a framework.

See Definition 4-13.

Class, extension-point

An extension-point class is a framework class that may be subclassed by framework-external classes.

See Definition 4-10.

Class, use-client

A use-client class of a framework is the class of a use-client object. It is connected to the framework through one or more role models.

See Definition 4-3.

Class set, built-on

The built-on class set of a framework is the set of all built-on classes of the framework.

See Definition 4-9.

Class set, extension-point

The extension-point class set of a framework is the set of all extension-point classes of the framework.

See Definition 4-11.

Class model

A class model is a set of classes and a set of role models. The classes relate to each other by inheritance and object relationship descriptions between role types. The class relationship graph must be non-partitioned.

See Definition 3-20 and 3-11.

Composition function

A composition function composes types. It is used as part of a class definition, where it composes the role types of a class to form the class type.

See page 37.

Framework

A framework is a class model, together with a free role type set, a built-on class set, and an extension-point class set.

See Definition 4-1.

Framework extension

A framework extension is a set of classes. Each class is either an extension class of the framework or a class that is transitively connected with at least one extension class through a role model.

See Definition 4-14.

Framework extension, domain-specific

A domain-specific framework extension is a framework extension that is not a framework, but that can be used by different applications in the same domain.

See Definition 4-15.

Framework extension, application-specific

An application-specific framework extension is a framework extension that is not a framework and that can be used by one specific application only.

See Definition 4-16.

Inheritance

An inheritance is a pair of classes (X, Y) such that any instance of class Y can be substituted in a context where an instance of class X is expected.

See Definition 3-9.

Object

An object is an opaque runtime entity of a system that provides state and operations to query and change that state. An object has a lifecycle: It is created, may change over time, and is possibly deleted. Objects can be identified unambiguously; identity is an intrinsic property of every object.

See Definition 3-1.

Object, built-on

A built-on object of a framework is a framework-external object that a framework object makes use of in an object collaboration task.

See Definition 4-7.

Object, use-client

A use-client object of a framework is a framework-external object that makes use of one or more framework objects in an object collaboration task.

See Definition 4-2.

Object aggregation

An object aggregation is a pair of objects (x, y), stating that an object x aggregates an object y as a part of it. To aggregate an object means to control it, not only to make use of it, but to determine its lifetime and accessibility as well.

See Definition 3-7.

Object aggregation description

An object aggregation description is a pair of types (X, Y) that determines possible runtime object aggregations. An aggregation between two objects (x, y) conforms to the aggregation description if x is of type X or a subtype of X, and if y is of type Y or a subtype of Y.

See Definition 3-8.

Object association

An object association is a pair of objects (x, y), stating that an object x holds a reference to another object y of which it may or may not make use.

See Definition 3-5.

Object association description

An object association description is a pair of types (X, Y) that determines possible runtime object associations. An association between two objects (x, y) conforms to the association description if x is of type X or a subtype of X, and if y is of type Y or a subtype of Y.

See Definition 3-6.

Object collaboration

An object collaboration is a set of objects that relate to each other by object relationships. An object collaboration is said to be valid if it conforms to a class model.

See Definition 3-10.

Object collaboration task

An object collaboration task is an object collaboration and a set of roles objects play in the collaboration. The object relationship graph must be non-partitioned.

See Definition 3-17.

Object system.

An object system is an object collaboration.

See page 31.

Role

A role is an observable behavioral aspect of an object.

See Definition 3-12.

Role constraint

A role constraint is a value from the set {role-implied, role-equivalent, role-prohibited, role-dontcare}. For every given pair of role types (R, S) from a role model one such value is defined.

See Definition 3-19.

Role-dontcare constraint

A role-dontcare value for a pair of role types (R, S) defines that an object playing a role r of role type R has no constraints with respect to another role s of role type S within the given collaboration task. The role s may or may not be available together.

See page 38.

Role-equivalent constraint

A role-equivalent value for a pair of role types (R, S) defines that an object playing a role r defined by role type R is always capable of playing a role s defined by role type S, and vice versa. That is, role r and role s imply each other. This relationship is symmetric and transitive.

See page 38.

Role-implied constraint

A role-implied value for a pair of role types (R, S) defines that an object playing a role r defined by role type R is always capable of playing a role s defined by role type S. That is, role r implies role s. This relationship is transitive.

See page 38.

Role-prohibited constraint

A role-prohibited value for a pair of role types (R, S) defines that an object playing role r defined by role type R may not play role s defined by role type S within a given collaboration task. That is, role r prohibits role s for the task. This relationship is symmetric and transitive.

See page 38.

Role model

A role model is a set of role types that relate to each other by object relationship descriptions and role constraints. The role type relationship graph must be non-partitioned.

See Definition 3-18.

Role model, free

A free role model of a framework is a framework-defined role model that has one or more free role types.

See Definition 4-5.

Role type

A role type is a type that defines the behavior of a role an object may play. It defines the operations and the state model of the role, as well as the associated semantics.

See Definition 3-13.

Role type, callback

A callback role type is a free role type of a framework that has a non-empty set of operations. It may be picked up by higher-layer classes. Callback role types are the role modeling equivalent of callback interfaces as used by traditional coupling mechanisms.

See Definition 4-12.

Role type, free

A free role type of a framework is a role type of a framework-defined role model that may be picked up by use-client classes by putting it into their role type sets.

See Definition 4-4.

Role type, no-operation

A no-operation role type is a role type that defines no operations.

See Definition 3-14.

Role type, no-semantics

A no-semantics role type is a no-operation role type that defines neither state nor behavior.

See Definition 3-15.

Role type set, free

The free role type set of a framework is the set of all free role types of a framework.

See Definition 4-6.

Value

A value is an atomic entity from the abstract and invisible universe of values. A value cannot be perceived directly, but only through occurrences of its representations. The representations are interpreted by means of interpretation functions. These interpretation functions return further (occurrences of representations of) values; they do not change the value.

See Definition 3-3.

Value type

A value type is a type that specifies a set of values together with the interpretation functions applicable to representations of members of this set.

See Definition 3-4.

C

Notation Guide

This appendix describes the graphical notation used in the diagrams of this dissertation. The notation uses diagrammatic UML syntax where possible, and adds to it where necessary.

C.1 Classes and role types

A class is depicted as a rectangle, with the class name set in bold font at the top of the rectangle. If the class name is set in Italics, the class is abstract. Class and class type is used synonymously.

A role type is depicted as an oval. The name of the role type is prominently centered in the oval. Set below it, in parentheses and a smaller font, is the name of the role model the role type is defined by.

Figure C-1 depicts an example class and several example role types. Here, a class ResourceService provides the three role types Service, Provider, and Singleton of the role models ResourceService and RSSingleton, respectively. Thus, the qualified name of the role types is ResourceService.Service, RSSingleton.Provider, and RSSingleton.Singleton.

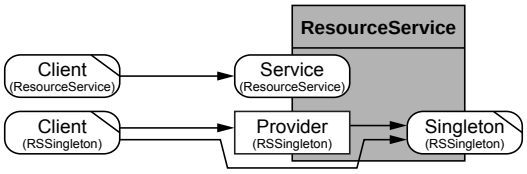

Figure C-1: Example class with role types.

Figure C-1 lets us distinguish between instance-level role types and class-level role types. An instance-level role type (a “regular” role type) is depicted as the aforementioned oval. An instance-level role type defines behavior that *instances* of a class conform to. A class-level role type is depicted as a rectangle (RSSingleton.Provider in Figure D-1). A class-level role type defines behavior that the *class object* representing the class conforms to.

Finally, Figure D-1 lets us distinguish between role types that have operations and role types that have no operations (still, expected behavior may be defined for these no-operation role types). A no-op role type is depicted with a gray mark in the upper right corner. Examples are the ResourceService.Client, the RSSingleton.Client, and the RSSingleton.Singleton role types. This property applies to instance-level and class-level role types alike.

Classes may relate to each other using class inheritance. Figure C-2 shows how the ResourceService class inherits from an abstract Service class (is a subclass of it).

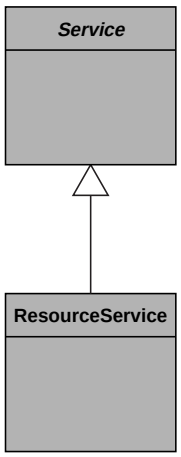

Figure C-2: Class inheritance.

The arrow inheritance symbol in Figure D-2 is taken from UML. See Chapter 3 for a discussion of the semantics of class inheritance in the context of role modeling.

C.2 Object relationships

Object relationship descriptions are relationships between types that constrain how objects conforming to these types may reference each other at runtime. Object relationship descriptions only relate the same kind of types with each other: class types with class types and role types with role types. Also, when connecting role types, object relationship descriptions are constrained to connect only role types of the same the role model.

Object relationship descriptions are depicted as connections between types. All relationship descriptions can be annotated with further information like direction and cardinality. Their meaning is taken from UML.

Figure C-3 shows three object relationship descriptions between role types as they may occur in a role model.

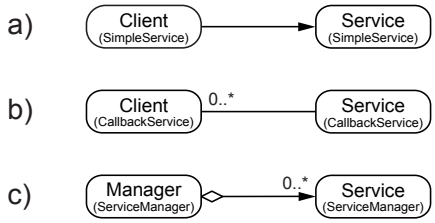

Figure C-3: Object relationships in role models

These three cases have the following meaning:

a) The arrow between the Client role type and the Service role type depicts a unidirectional use-relationship between an object playing the Client role and an object playing the Service role.

b) The line between the Client role type and the Service role type depicts a bi-directional use-relationship between objects, each of which plays one of the roles.

c) The diamond at the start of the arrow indicates ownership of the object pointed at, and the star at the end of the arrow indicates a cardinality of many (following UML rules).

The scoping of the object relationship descriptions by a role model or class model is important. No role type may relate to a role type from another role model by an object relationship description. See Chapter 3 for a discussion of how classes and role types relate to each other with respect to object relationship descriptions.

C.3 Class and role models

A class model is a set of interrelated classes, tied together by class inheritance and object relationship descriptions. A role model is a set of role types, tied together by role constraints (see below) and object relationship descriptions.

Figure C-4 shows three example role models, one binary, two ternary. The role model name is set below the role type name. The role model name qualifies the role type name so that role types with the same name, for example the three Client role types, can be distinguished.

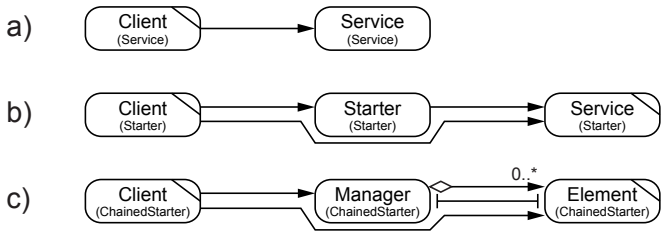

Figure C-4: Three example role models.

Figure C-5 shows one class model, consisting of three classes, and three example role models (from above).

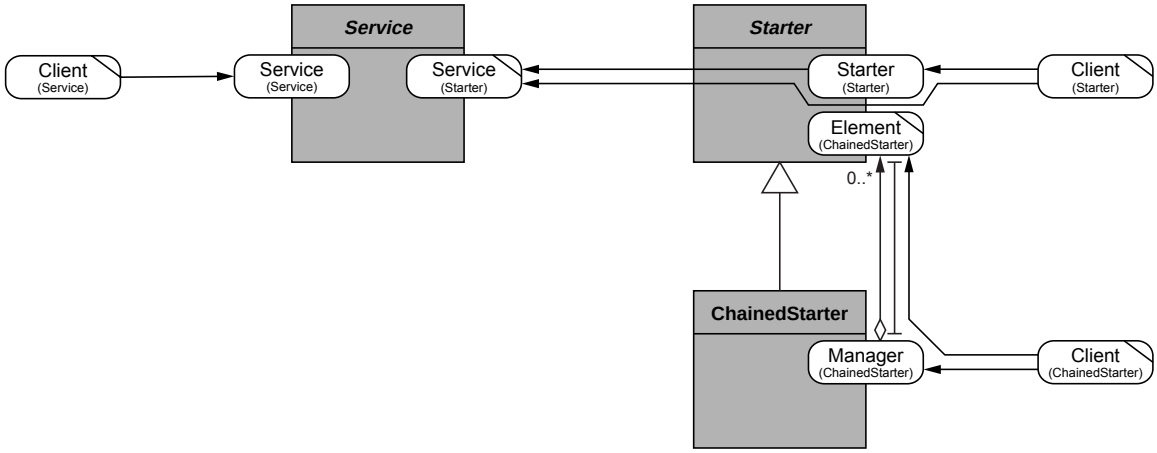

Figure C-5: Example class model.

The example class model does not show its free role types. Free role types are color-coded with a gray background (see the section on frameworks below).

C.4 Role constraints

Role constraints are values that define how roles played according to two role types from the same role model may come together in one object. For any given pair of role types, there is one role constraint value. The default value is role-dontcare (see below).

Role constraints are depicted as connections between role types. Figure C-6 shows the four different role constraints that may occur in a role model.

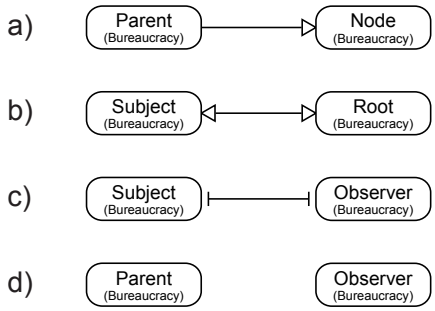

Figure C-6: Role constraints in a role model.

These four cases have the following meaning:

a) *Role-implied.* The white-headed unidirectional arrow depicts a role-implied role constraint. The role constraint value of (Parent, Node) is role-implied, the role relationship value of (Node, Parent) is role-dontcare.

b) *Role-equivalent.* The white-headed bi-directional arrow depicts a role-equivalent role constraint. The role constraint value of both (Subject, Root) and (Root, Subject) is role-equivalent.

c) *Role-prohibited.* The bar-headed bi-directional arrow depicts a role-prohibited role constraint. The role constraint value of both (Child, Root) and (Root, Child) is role-prohibited.

d) *Role-dontcare.* The missing of a symbol depicts a role-dontcare role constraint. The role constraint value of both (Client, RootClient) and (RootClient, Client) is role-dontcare.

Role constraints connect role types within one role model. A role-prohibited constraint is scoped by the runtime object collaboration task. It is possible for an object to play roles according to two different role types (even if there is a role-prohibited constraint between them) given that these roles are played in different role model instances, that is different object collaboration tasks.

C.5 Role model shorthands

Some role model compositions keep recurring throughout the examples. The primary examples are Object Creation and Singleton Access. Therefore, the dissertation uses shorthands to conveniently represent these recurring compositions as one single role model. Such a shorthand is much like a template in that it needs to be applied and adapted to a specific situation.

Figure C-7 shows an example of the applied Object Creation shorthand, here for the creation of a ResourceService instance.

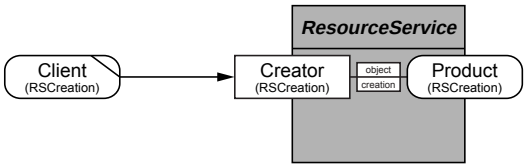

Figure C-7: Object Creation shorthand, applied to the ResourceService example.

Figure C-8 shows the expanded form of the Object Creation shorthand.

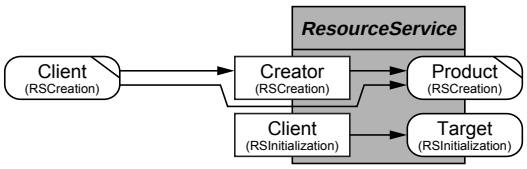

Figure C-8: Expanded form of the applied Object Creation shorthand.

Figure C-9 shows an example of the Singleton Access shorthand, here for the access to a single ResourceService instance.

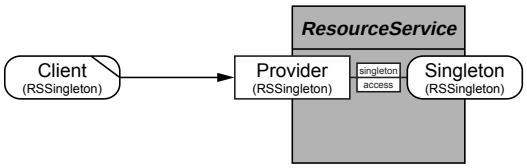

Figure C-9: Singleton Access shorthand, applied to the ResourceService example.

Figure C-10 shows the expanded form of the Singleton Access shorthand.

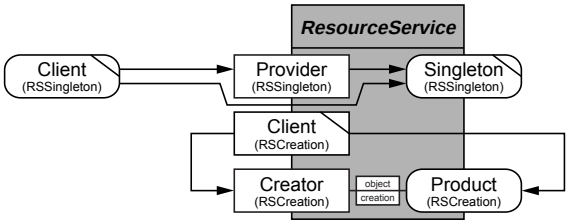

Figure C-10: Expanded form of the Singleton Access shorthand.

Another candidate for a shorthand is the composition of a Manager role model, in which a Manager object maintains a dictionary of Elements, with the lookup functionality of a dictionary and its key objects.

Composite (compound) pattern applications [Rie97c] are not a suitable subjects for shorthands, because it is important to see the constituting role models. Shorthands should be used only for trivial role model compositions.

If non-trivial role model compositions are needed frequently and lead to cluttering up the figures, an explicit semantic construct for composing role models can be introduced and given a diagrammatic representation.

C.6 Frameworks

Frameworks are class models with well-defined boundaries. A visual border with the framework's name attached to it surrounds the framework classes. Also, free role types of the framework are color-coded in gray.

Figure C-11 shows a simple framework based on the Service and Starter class model from above.

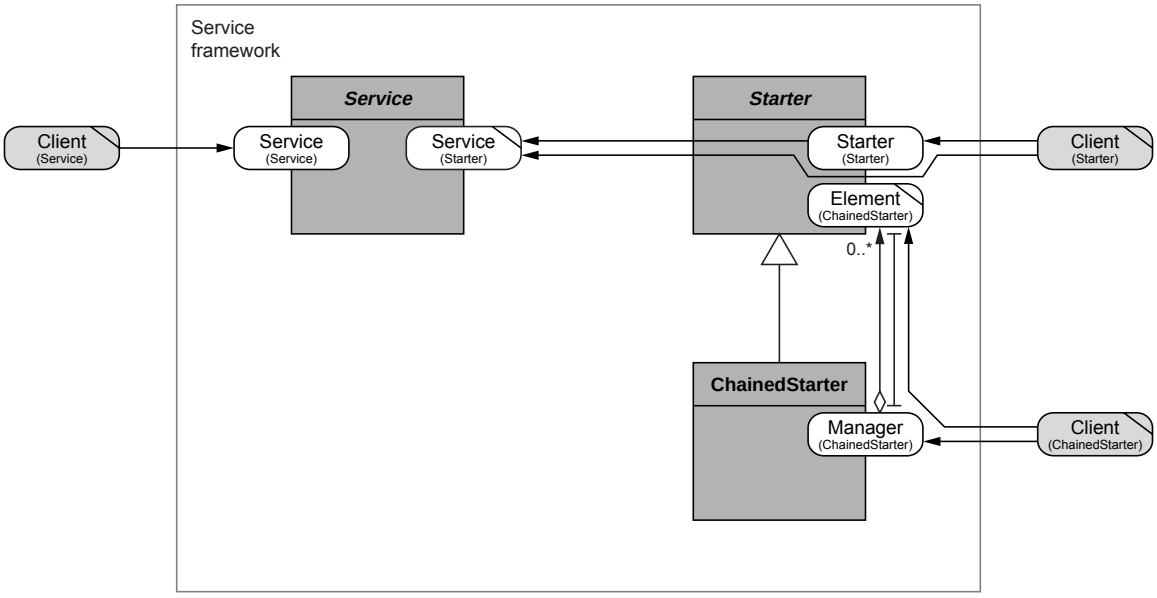

Figure C-11: Example framework.

The framework comprises the Service, Starter, and ChainedStarter classes. It offers three free role types for client classes to pick up: Service.Client, Starter.Client, and ChainedStarter.Client. The color-

coding of these free role types is maintained, even if they are assigned to classes in a client context, because they are still free for other clients to pick them up.

D

Design Patterns

This appendix describes several design patterns that are used in the main body of the dissertation using a role model form. The patterns are essentially the same as found in their original documentation, except that the role model form portrays them in a different light. The use of role models lets us more flexibly assign role types to classes than the class-based form allows us to do. While the class diagrams of the patterns frequently might look simpler than the role model diagrams, this is not the case. The complexity of a pattern is always the same, independently of its presentation form.

The pattern descriptions in this appendix have the following properties:

- The descriptions of the patterns are incomplete. They merely serve as a reminder for those who forgot or do not know a specific pattern under the given name so they can quickly look it up. For a more detailed description, references are provided.
- The patterns are illustrated rather than rigidly defined. There is no design pattern notation behind the pattern descriptions except than an intuitive understanding about what the role model illustration might communicate to developers regarding the pattern instantiation.
- None of the descriptions is to be taken as a rigid definition. (See the discussion of pattern vs. template in Chapter 3). Role models are more flexible than class model and suggest a wider application [Rie96a], but even they cannot encompass all possible design templates.

In the diagrams, some role types and object relationship descriptions are grayed out. These role types are required in an instantiation of a pattern, but are not considered to be an integral part of the core pattern role model illustration. The primary example of such a grayed-out role model is the role type pair (Client, Object) that simply states that an object provides some domain functionality to a Client.

D.1 Abstract Factory

The Abstract Factory pattern centralizes the creation of objects from a family of products in a Factory object. A Client requests new Product objects from a Factory object. The Factory object ensures consistency among a family of Product objects and hides the details of the object creation process.

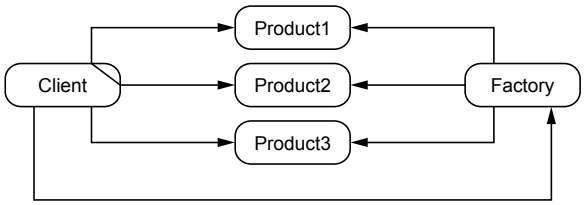

Figure D-1: Role model of the Abstract Factory pattern.

A class-based description of this pattern can be found in [GHJV95].

D.2 Adapter

The Adapter pattern adapts an existing object to a new use-context by means of an intermediate Adapter object. A Client uses the Adapter operations only, and the Adapter implements them in terms of the domain functionality of the Adaptee.

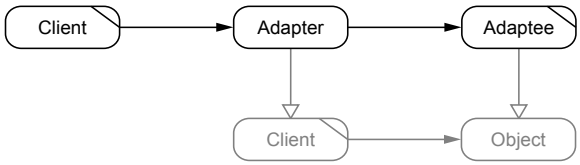

Figure D-2: Role model of the Adapter pattern.

A class-based description of this pattern can be found in [GHJV95].

D.3 Bridge

The Bridge pattern splits the implementation of a domain concept into an Abstraction and an Implementor object so that both can be varied independently. The Abstraction provides the primary domain functionality, and the Implementor provides the implementation primitives all variations of the Abstraction can be implemented by. The Client makes use only of the Abstraction. The Abstraction owns its Implementor, uses its operations for its own implementation, and hides it from the Client.

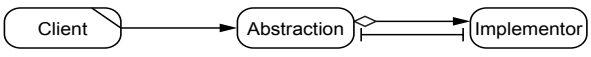

Figure D-3: Role model of the Bridge pattern.

A separate Client (not shown in the figure) configures the Abstraction with its Implementor. A common type of Client is an Abstract Factory that returns a preconfigured Bridge upon Client request.

A class-based description of this pattern can be found in [GHJV95].

D.4 Chain of Responsibility

The Chain of Responsibility pattern determines the target object of a client request dynamically by passing the request along a chain of objects. Each object in the chain may decide whether to execute, drop, or pass on the request. A Predecessor forwards the request to its Successor.

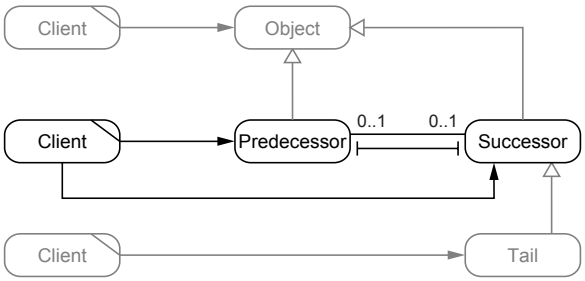

Figure D-4: Role model of the Chain of Responsibility pattern.

Using an object chain this way lets us configure the recipient of the request dynamically. A separate client configures the chain of objects.

A class-based description of this pattern can be found in [GHJV95].

D.5 Class Object

The Class Object pattern provides functionality common to all objects of a class in one Class object. A Class object can be asked for meta-information about any of its Instance objects. In contrast to a Type object, the Class object provides not only operations to inspect its Instances and provide information about it, but also functionality to create and change its Instances.

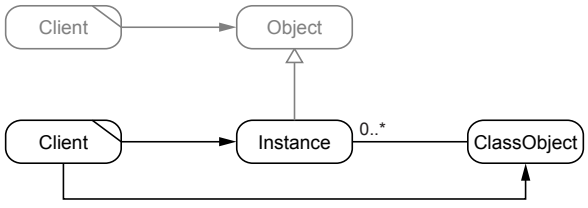

Figure D-5: Role model of the Class Object pattern.

Type Object is one element of the pattern triple Metaobject, Type Object, and Class Object. The distinction between Class Object and Type Object is done pragmatically. Class Objects provide implementation information about its Instances and they can manipulate and create Instances. Type Objects

provide application domain specific information rather than implementation information; their implementations may be heterogeneous, and they cannot manipulate their Instances.

To my knowledge, there is no commonly known class-based description of the pattern. However, any major object-oriented system provides an implementation of this pattern.

D.6 Composite

The Composite pattern determines how to build a hierarchy of objects. Any object in the hierarchy is a Child, or a Parent, or both. A Child may receive a Parent object, and a Parent object may receive or drop some Child objects. The Child and Parent role types serve to configure and maintain the hierarchy. A Client configures a Parent with its Child objects.

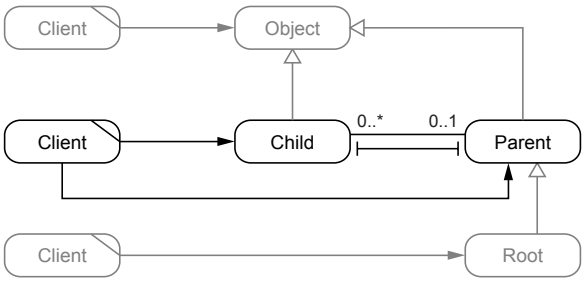

Figure D-6: Role model of the Composite pattern.

A class-based description of this pattern can be found in [GHJV95].

D.7 Decorator

The Decorator pattern lets us transparently add functionality to an existing object through object composition. A Core object is wrapped by a Decorator object. The Client makes use of both the Core and the Decorator without seeing to different objects.

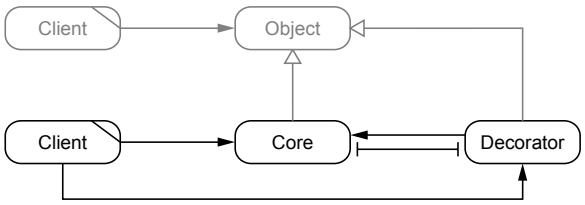

Figure D-7: Role model of the Decorator pattern.

A class-based description of this pattern can be found in [GHJV95].

D.8 Factory Method

The Factory Method pattern puts the creation of an object in method of its own that can be varied independently from the Client using the method. The Factory Method is provided by a Creator object that returns a Product object upon Client request.

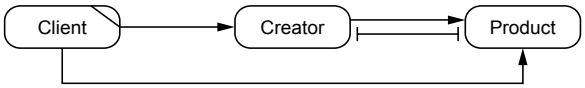

Figure D-8: Role model of the Factory Method pattern.

A class-based description of this pattern can be found in [GHJV95].

D.9 Manager

The Manager pattern puts the management of some Elements into a Manager object so that Element management gets independent of the Client and the Elements. Clients request Elements from the Manager. The Manager owns the Elements. It creates, provides, and deletes them.

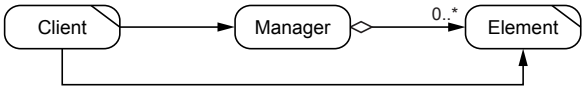

Figure D-9: Role model of the Manager pattern.

In [Som98], Sommerlad describes a class-based variant of this pattern, also called Manager. Sommerlad's Manager requires the set of Elements to be homogeneous, while the definition of Manager here accepts a heterogeneous collection of Elements.

D.10 Mediator

The Mediator pattern centralizes the communication of a set of Colleague objects in one Mediator object. The Colleagues do not communicate with each other directly, but only through the Mediator. This reduces communication complexity from square(n) to n. It facilitates the introduction and removal of a new Colleague object without affecting the other Colleagues.

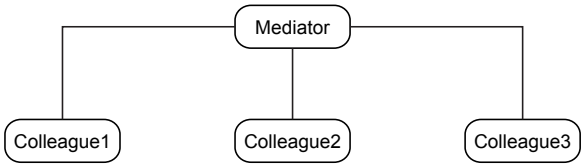

Figure D-10: Role model of the Mediator pattern.

A class-based description of this pattern can be found in [GHJV95].

D.11 Metaobject

The Metaobject pattern separates the domain-specific functionality of an object from the technical procedure of executing this domain functionality. Clients send requests to the Metaobject for execution on a specific BaseObject. The Metaobject defines the procedures for executing incoming requests, and the BaseObject provides the functionality to invoke the domain-specific operations. The Metaobject typically deals with issues like request queuing and synchronization, and the BaseObject provides a dynamic invocation interface for calling the domain-specific operations.

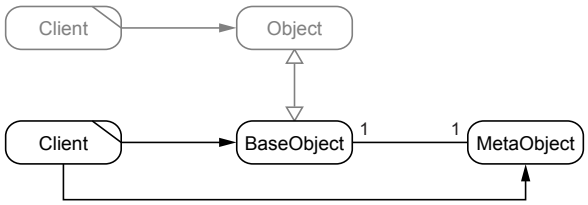

Figure D-11: Role model of the Metaobject pattern.

The configuration of the Metaobject can be complex (see Chapter 6 for an example). Metaobjects are part of a metalevel architecture that represents the overall (conceptual and/or technical) framework for handling metalevel issues.

Metaobject is one of the pattern triple Metaobject, Type Object, and Class Object.

To my knowledge, there is no commonly known class-based description of the pattern. However, every object-oriented system based on an explicit metalevel architecture is likely to feature an instance of this pattern.

D.12 Null Object

The Null Object pattern serves to provide a null implementation of a domain concept. The null implementation provides null behavior, which is the behavior assumed to be executed if no object were present at all. The pattern lets developers set an object reference to the null object rather than to a null reference and avoids cluttering the client code with checks whether the reference is null or not.

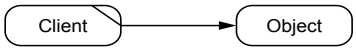

Figure D-12: Role model of the Null Object pattern.

In [Woo98], Woolf describes the Null Object pattern. Because it is only about implementation, the pattern cannot be expressed well using role modeling.

D.13 Object Registry

The Object Registry pattern centralizes access to a set of Element objects in one Registry object. Clients register and unregister Elements at the Registry giving them convenient names for later retrieval. Such a Registry is typically a thread-local or process-local Singleton.

Figure D-13: Role model of the Object Registry pattern.

The Object Registry pattern is to be distinguished from the Value Registry pattern (not documented here). An Object Registry handles objects, and a Value Registry handles values. Clients of an Object Registry have to be aware of possible side-effects, while clients of a Value Registry do not have to do so.

D.14 Observer

The Observer pattern decouples a set of Observers from a Subject. It is used to maintain state dependencies between the Observers and their Subject. In case of a state change, the Subject sends out an event to notify its Observers about the change. The Subject does not rely on any specific type of Observer, but uses a common and minimal Observer protocol only. A Client configures the Subject with its Observers (Client and Observer object may well be the same).

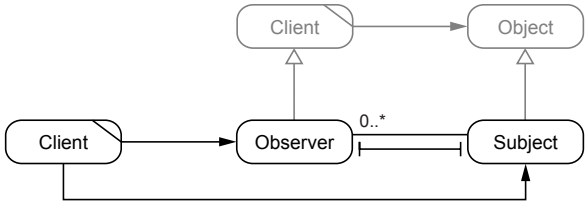

Figure D-14: Role model of the Observer pattern.

In Java, the Observer pattern takes on the form of EventListeners.

A class-based description of this pattern can be found in [GHJV95].

D.15 Product Trader

The Product Trader pattern separates the creation of a Product object from the Client requesting it by putting the creation process into a Trader object. Clients request new Products from the Trader using a specification of the Product (rather than naming its class). The Trader uses the specification to select an element from a set of Elements. Each of the Elements can act as a Creator for the Product.

When asked for a Product, the Trader selects an Element based on the specification provided by the Client. The Trader then acts as a Client of the Element that acts as the Creator for the new Product

object. The Trader/Client object asks the Element/Creator object for a new Product, which is then returned to the Client.

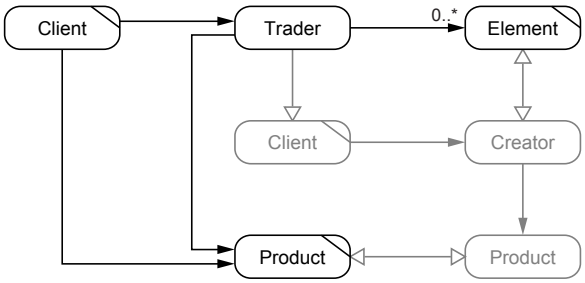

Figure D-15: Role model of the Product Trader pattern.

In [BR98], Bäumer and Riehle present a class-based description of this pattern.

D.16 Property List

The Property List pattern makes a Provider object provide a generic and extensible set of Properties to Clients. A Client asks a Provider object about its Properties. Properties are accessed using a naming scheme, for example simple strings, and generic get and set operations rather than through property-specific operations. The Provider defines which Properties it offers. Every implementation of a Provider may provide its own set of Properties. Properties may even be added and removed at runtime.

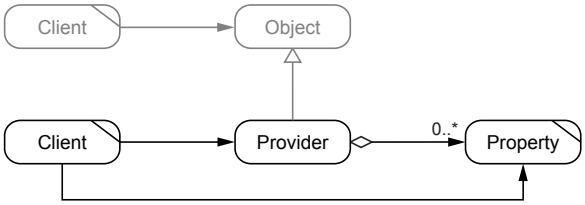

Figure D-16: Role model of the Property List pattern.

In [Rie97a], Riehle describes the pattern in more detail using role modeling.

D.17 Prototype

The Prototype pattern lets Clients create complex Product objects by cloning a Prototype object. Products are copies of the Prototype and reflect its possibly complex object configuration. Clients request a copy of the Prototype, which they receive as a new Product object. Prototypes serve as representatives of one or several complex objects and can be handled generically.

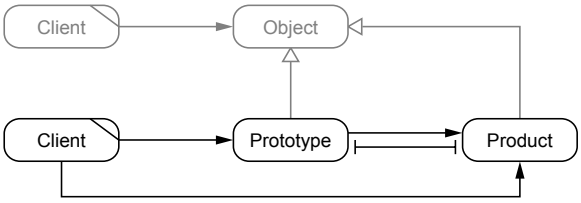

Figure D-17: Role model of the Prototype pattern.

A class-based description of this pattern can be found in [GHJV95].

D.18 Role Object

The Role Object pattern transparently attaches Role objects to a Core object. The Core represents an important domain concept, and the Roles represent some domain-specific extension of the Core concept. Clients make use both of the Roles and the Core. The Core manages its Roles and provides them to a Client upon request. Roles may be retrieved from the Core using a simple naming scheme, for example strings.

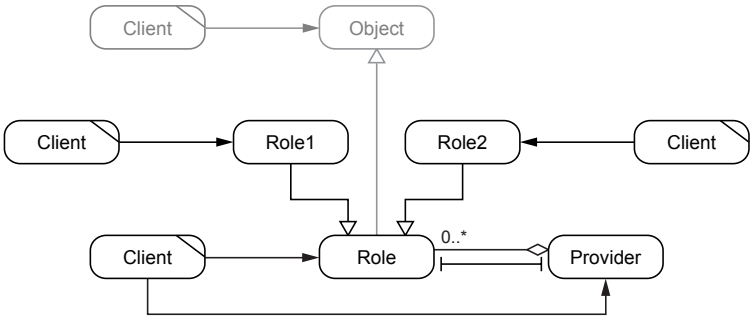

Figure D-18: Role model of the Role Object pattern.

The Core may be configured with Role objects in many different ways, for example during configuration time from configuration data or during execution time through dedicated clients.

In [BRSW00], Bäumer et al. present a class-based description of this pattern.

D.19 Type Object

The Type Object pattern centralizes common information about a set of Instance objects in a Type Object that is shared by all Instances. Clients ask the Type Object of an Instance for information about the Instance. This information may also be directly provided by an Instance, but it will be implemented then by asking the Type Object. Using Type Objects, otherwise redundant information about common properties of all Instances is provided in a single place, the Type Object.

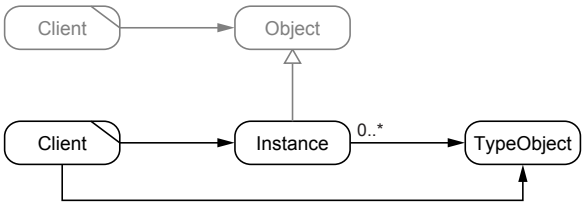

Figure D-19: Role model of the Type Object pattern.

Type Object is one of the pattern triple Metaobject, Type Object, and Class Object. The distinction between Class Object and Type Object is done pragmatically. Class Objects provide implementation information about its Instances and they can manipulate and create Instances. Type Objects provide application domain specific information rather than implementation information; their implementations may be heterogeneous, and they cannot manipulate their Instances.

In [JW98], Johnson and Woolf present a class-based description of this pattern.

D.20 Serializer

The Serializer pattern reads Readable objects from a Reader and writes Writable objects to a Writer. The Reader reads the object information from a specific backend, and the Writer writes the object information to a specific backend. Backends vary with Reader/Writer implementations. The Serializer pattern is used to serialize objects for different purposes like making them persistent, marshalling and unmarshalling them, and debugging them.

Readable and Reader as well as Writable and Writer objects collaborate recursively. A Readable reads all of its attributes from a Reader. For attributes that are Readable object references, the Reader to creates the Readable and then tells it to read its attributes from it, the Reader. Similarly, a Writable writes its attributes to a Writer that in turn tells a Writable attribute to write its attributs on it, the Writer. Primitive value types like integer and string attributes end the recursive descent.

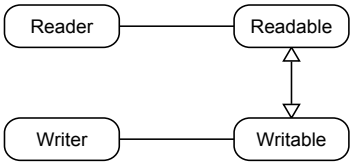

Figure D-20: Role model of the Serializer pattern.

The Serializer pattern can be viewed as the repeated specialized composition of the Visitor pattern.

In [RSB+98], Riehle et al. present a class-based description of this pattern.

D.21 Singleton

The Singleton pattern serves to ensure that there is exactly one instance of an object, the Singleton, in a given operation context, and to provide a central convenient access point to it. Historically, the op-

eration context is the process, but it could be a thread as well. A Client requests the Singleton from a Provider. If necessary, the Provider creates the Singleton on demand.

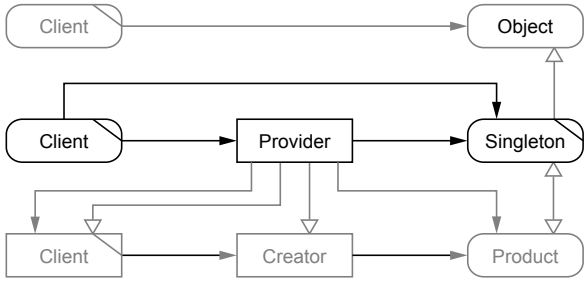

Figure D-21: Role model of the Singleton pattern.

Because the combination of an applied Singleton role model with an object creation role model of Client, Creator, and Product role types ocurrs frequently, the dissertation uses a shortcut for it.

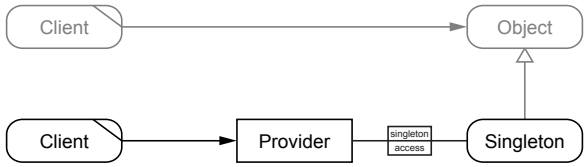

Figure D-22: Shortcut role model of the Singleton pattern.

A class-based description of this pattern can be found in [GHJV95].

D.22 Specification

The Specification pattern provides descriptive elements about an object to a client for use in object selection based on specifications. A Client requests a Specification from a Provider. The Specification describes one or several properties of the Provider. Typically, the Specification can provide a unique key to a client that is computed based on the properties described by the Specification and that distinguishes the Specification from Specifications of other Providers.

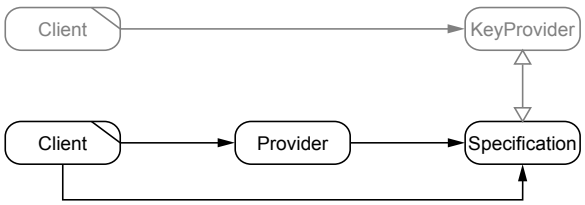

Figure D-23: Role model of the Specification pattern.

In [Rie96c], Riehle describes the use of the Specification pattern in the context of class selection and object creation (also known as trading), and in [EF97], Evans and Fowler describe several patterns that show how to define and compose complex specifications.

D.23 State

The State pattern serves to split a large state space of an object into several distinct parts to ease the object's implementation, to manage the state space more easily, and to extend it more easily. An Object providing domain functionality to a Client acts as the Context for a set of State objects. The Context forwards Client requests to one State object, which implements the requested behavior.

At any given time, exactly one State object is active, representing the subspace of the overall state space of the object the current state vector is in. If state changing operations cause the state vector to leave the current subspace, another State object becomes active, representing the correct subspace. A State object implements the behavior according to the rules of the subspace it represents.

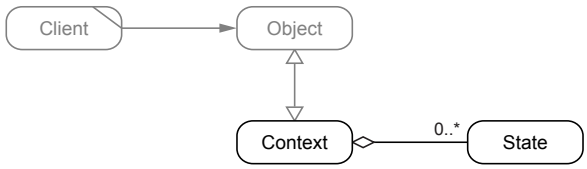

Figure D-24: Role model of the State pattern.

A class-based description of this pattern can be found in [GHJV95].

D.24 Strategy

The Strategy pattern serves to configure a domain object with an instance from a family of algorithms, rather than hard-coding any specific algorithm in the domain object. The Strategy object encapsulates the algorithm, and is set to its Context by a Client. Whenever the domain object has to execute the algorithm it acts as the Context of the Strategy and delegates the task of performing the algorithm to it.

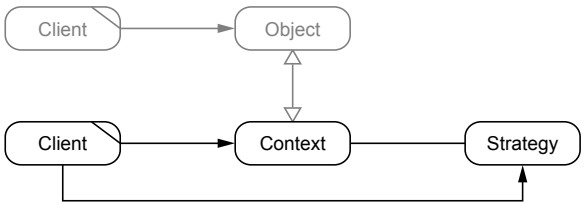

Figure D-25: Role model of the Strategy pattern.

A class-based description of this pattern can be found in [GHJV95].

D.25 Visitor

The Visitor pattern serves to extend an existing object structure with new external algorithms. The different object types from the structure are represented as different Node role types. Common to all objects is the Element role type. A Visitor object represents a new external algorithm. For each of its Node type attributes, the Element dispatches on a Visitor for the particular Node type. The Visitor can then execute the behavior associated with that particular Node type.

Because Node objects can always act as Elements, a Visitor may recursively descent into the object structure (if it is a hierarchy). An Element dispatches on an a Visitor for a given Node type attribute, and the Visitor calls on the Node type attribute to dispatch back to the Visitor on the Node type's attributes. Primitive value types and objects that are not Elements end the recursive descent.

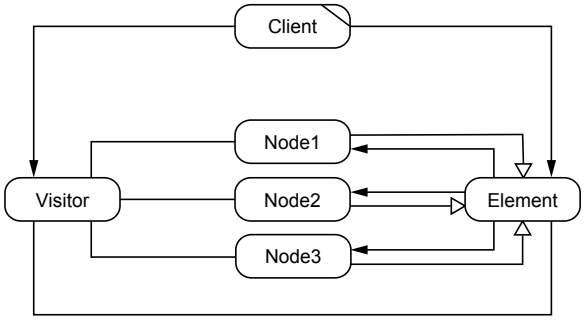

Figure D-26: Role model of the Visitor pattern.

A class-based description of this pattern can be found in [GHJV95].

E

Pointers to Further Material

This appendix provides pointers to information referred to in this work.

The following page provides the most recent pointers to all of the materials listed below:

- Index page: http://www.riehle.org/diss/index.html

At the time of writing this dissertation, the original information can be found at:

- JHotDraw 5.1: http://members.pingnet.ch/gamma/JHD-5.1.zip
- JHotDraw 5.1 tutorial: http://www.eos.dk/jaoo/presentations/index.html

An index to my own publications referenced in Appendix A can be found at:

- Publication index: http://www.riehle.org/papers/index.html

Curriculum Vitae

Name: Dirk Riehle
Nationality: xxxxxxxxx
Date of birth: xxxxxxxxx

xxxx-xxxx: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.
xxxx-xxxx: Matthias-Claudius-Gynmasium (high school), Hamburg, Germany.
1988-1991: University of Hamburg, Germany. Physics studies, Vordiplom.
1988-1995: University of Hamburg, Germany. Computer science studies, Dipl.-Inform.
1995-1996: UBS AG, Corporate Clients IT division, Zürich, Switzerland. Architect and developer.
1996-1999: UBS AG, Ubilab, Zurich, Switzerland. Researcher, architect, and developer.
1997-1999: ETH Zurich, Switzerland. Dissertation, supervised by Prof. Dr. Thomas R. Gross.
1999: Credit Suisse, Data Warehousing, Zurich, Switzerland. Software architect.
1999-current: SKYVA International, Cambridge, MA, and Mannheim, Germany. Software developer.