JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 3, May-June 2003

A Construct for Effectively Implementing Semantic Associations

Dr. Charles A. Suscheck, TurboPower Software, Colorado
Dr. Bo Sandén, Colorado Technical University, Colorado

Abstract

Associations are a key concept in object-oriented modeling. Implementing purely semantic associations with direct containment can lead to reduced cohesion and increased coupling as well as difficulties with referential integrity. Implementing semantic associations using the constructs shown in this paper will lead to domain objects that are more flexible and reusable. Adding container classes for domain objects and their association objects leads to a high level of traceability between the conceptual model and it's implementation.

1 THE IMPORTANCE OF ASSOCIATIONS

Object-oriented (OO) techniques were specifically developed in order to reduce domain complexity and communication between domain experts and systems developers can understand. OO models the user's perspective of the system in a semantically meaningful manner that follows human conceptualization. "Object-oriented systems allow the real world to be represented more directly than do conventional ones" [Gottlob96].

Associations play a key role in object-oriented domain modeling. They capture the nature of the domain by depicting relationships among objects. An association is a group of links between instances with common semantics and structure, the key point being an association involves a semantic relationship between two or more classes.

Associations can be either static or dynamic in nature. Dynamic interactions depict sending messages or signals between classes. Static structures such as inheritance, aggregation, and composition are also forms of associations.

A third category of associations is a purely semantic association. Semantic constructs such as roles and relationships comprise this group. Associations that support semantic constructs are easily modeled in UML, but inconsistently implemented. Semantic associations enrich the understanding of the system by capturing the nature of the domain and, in certain domain constructs, play a key role in the understanding of the system's conceptual model.

Cite this article as follows: Charles A. Suscheck, Bo Sandén: "A Construct for Effectively Implementing Semantic Associations", in *Journal of Object Technology*, vol. 2, no. 3, May-June 2003, pp. 101-111. http://www.jot.fm/issues/issue_2003_05/article1

Purely semantic associations are the focus of this article. Dynamic associations and static associations are well understood and implementation is handled through native constructs in nearly all object-oriented languages. Semantic associations are implemented inconsistently, if at all, in development languages which necessarily puts their implementation in the hands of the developer. We will focus on implementing semantic associations and how container classes play a key role in associations.

2 STRUCTURE OF ASSOCIATIONS

General Association Concepts

An association is by default bidirectional meaning that it can be read from either end with significance. An example is an "is_married_to" association between a class Man and a class Woman. Adding an arrowhead at one end specifies that the association is only *navigable* in one direction as in Fig. 1. Given a Radar object, the associated Beam objects can be identified, but a Beam object has no reference to the Radar object emitting it.

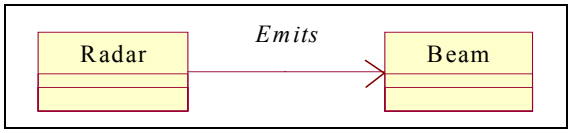


Figure 1: An association with direction

Associations can have other properties such as *multiplicity*, which constrains the number of related objects. In Fig. 2, a workstation displays data upon zero or more windows, but the data displayed on a given window comes from exactly one workstation.

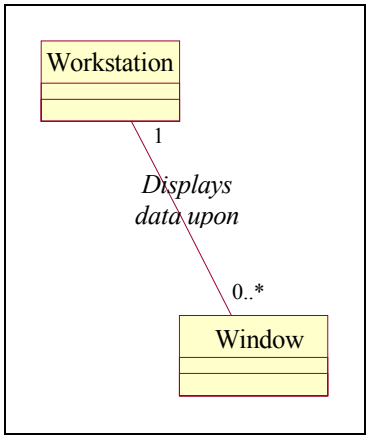
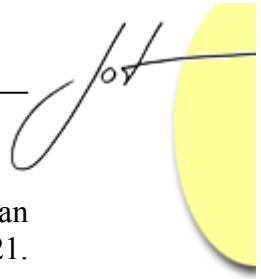


Figure 2: An association with multiplicity



Association constraints can be added to restrict the instances that can participate in an association. In Fig. 3 a club employs a bartender. The bartender must be older than 21. The association is constrained by the age of the bartender. Constraints can affect when an association is formed or what instances of a class can be associated.

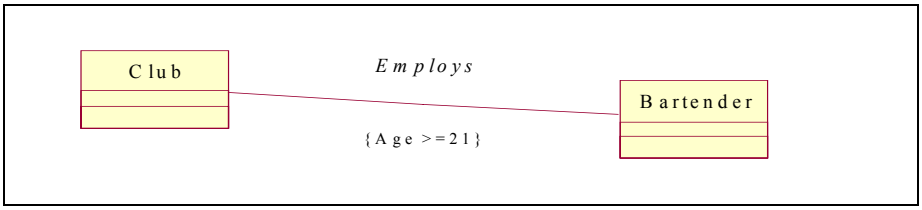


Figure 3: A constraint on an association.

Properties can be attached to an association by means of an *association class* [Rumbaugh91], [Booch99]. An association class has exactly one instance for each set of objects linked through the association and a lifetime delimited by the existence of the association. If a link is dissolved, the association class instance is destroyed. In Fig. 4, the radar detects a flying entity. Due to the association, certain information exists that is specific to the association, namely the DetectsParms class. The DetectsParms association class can contain such information as time of detection, radar cross section, and signal strength, which are only relevant when the radar detects the flying entity. In UML a dashed line is used to specify an association class.

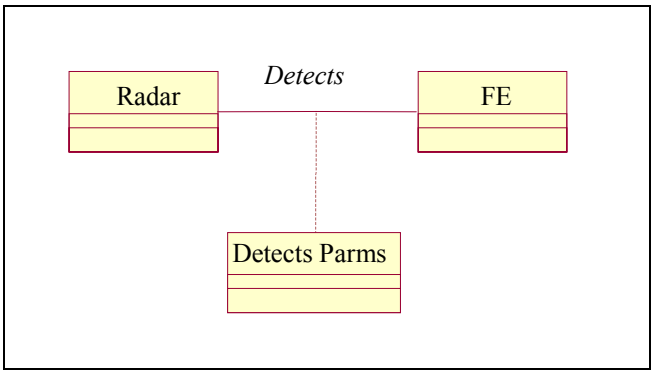


Figure 4: An association class, DetectsParms

Associations don't necessarily have to be binary, although the vast majority are. Associations between more than two instances (known as higher order or n-ary associations) are difficult to deal with [Rumbaugh91] from both an implementation standpoint and a cognitive standpoint. Fig. 5 shows an example of a ternary association. A person uses different computer languages on different projects.. Ternary associations cannot be subdivided without losing information, but can usually be replaced by binary associations if additional classes are introduced. In the example in Fig. 5 we can add a

class Job that is linked many - to one to each of Person, Project and Computer Language. Higher order associations are not to be confused with association classes in which the association class only exists for the duration of the association. The participants in n-ary associations are first class objects and can exists outside of the context of the association. A diamond is used in UML to specify an n-ary association [Rumbaugh99].

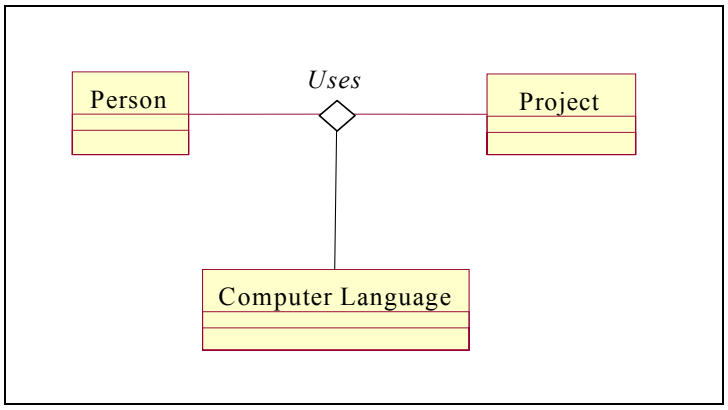


Figure 5: A ternary association from (Rumbaugh 1987)

Semantic associations

Semantic associations support relationships that are loose (not appropriate as direct or indirect containment) and are based on the semantics of the domain. Semantic associations are necessary of the understanding of the conceptual model, but require a loose coupling when moving to implementation.

Roles in associations are an example of a purely semantic relationship. A role is the function, behavior, or assigned characterization that an object plays in an association. For example, in Fig. 6, a person plays the role of employee while the company plays the role of employer in the works-for association. The actual association is “works-for”. Role names are sometimes used instead of association names when describing the association. Role names are particularly important when associating instances of the same class. There are very thorough discussions about roles in [Rumbaugh87], [Whitehurst97], [Kendall99].

Another form of purely semantic association is one that defines an important relational state. In Fig. 7 the Radar and the Flying Entity (FE) are related by a state association. The Radar detects the FE and the FE is detected by the Radar. They have a relationship that changes their state, yet the relationship is not a role, nor an association that produces a distinct physical implementation. The FE is now detected and the Radar enters a tracking state (a role based behavior).

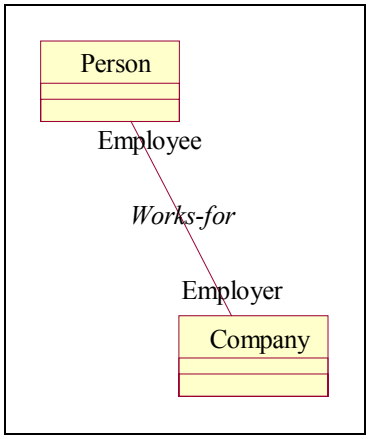


Figure 6: Roles in an association (Rumbaugh 1987)

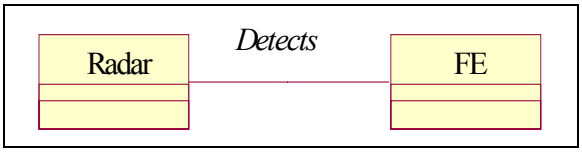


Figure 7: A loose state association

Researchers have introduced a number of complex constructs to support domain modeling [Kristensen94]. One of the more common complex constructs is *Role based behavior* [Whitehurst97]. The behavior of an object can change depending on the role it plays. When an association is formed between two instances, the behavior of the associated instances is altered in some way. A real world example is a person who becomes a parent. The person has a parental association with a young person (a child) and the behavior of the person is changed due to this association. Another example is a radar that has detected a flying entity. The radar now enters a tracking phase where it moves to follow the detected entity.

3 IMPLEMENTATION OF ASSOCIATIONS

Many tools that generate code from UML diagrams use a reference pointer to implement all associations. If a Radar emits a beam, it would contain an attribute Emits that holds a pointer to the Beam. This is exactly the same implementation scheme as containment by reference. The semantics of the directional association are lost and the loose coupling inferred by the conceptual model are lost. The Radar and Beam relate, but their relationship is not so much one of aggregation (like a bolt being contained in the Radar) as it is of a semantic construct that isn't captured by simply including a pointer in the Radar. Herein lies the problem: semantic associations are often implemented with tight coupling between the participants.

As pointed out in [Whitehurst97], associations should be treated as first-class entities. They should not be buried inside objects since they are not subordinate to a particular object but depend upon two or more classes. The information transcends a single class and should be treated as a first class entity. "Some information inherently transcends a single class, and the failure to treat associations on an equal footing with classes can lead to programs containing hidden assumptions and dependencies." [Rumbaugh91] .

Implement through direct containment

Associations can be implemented by means of one way or two way pointers (direct containment) or by additional object constructs [Odell95]. Fig 8 shows containment with two-way pointers. A diamond is used to indicate the root of a pointer. If a radar detects multiple flying entities, the Radar object contains an array of pointers.

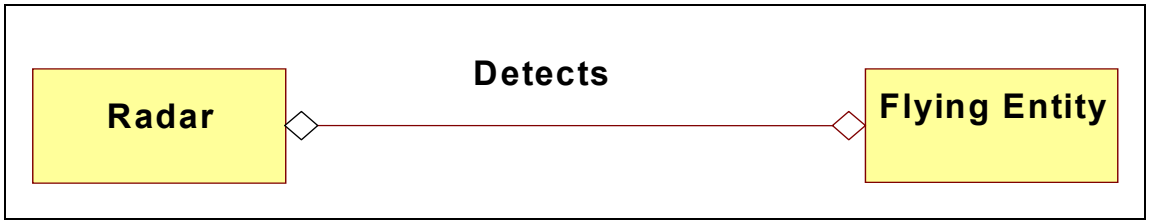


Figure 8: Using bi-directional containment to implement associations

Although easy to implement, direct containment has problems with extensibility. Adding a new association to a class means adding a new array of pointers. If an association is no longer viable, an empty pointer array is still contained within the class. Another problem is the potential for referential integrity rifts. In Fig. 8, if the radar is deleted, the pointer in the Flying Entity must be updated to no longer reference the radar.

Eventually, associations that are not pertinent to the scenario being modeled must be added to objects since a single object must have methods for *every* protocol that it can *possibly* participate in, not just those that are presently being used [Whitehurst97]. For example, in Fig. 9, a radar may have a number of association not necessarily pertinent to the semantics being described.

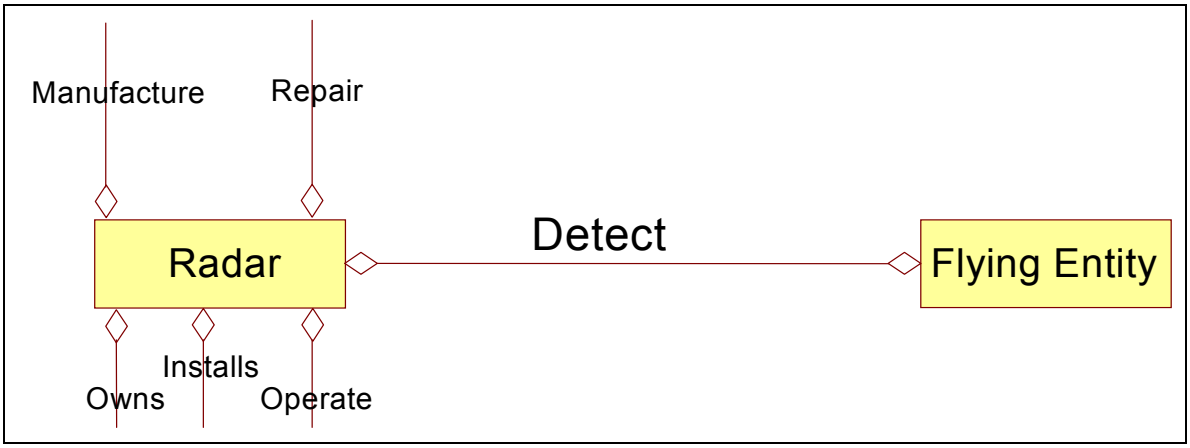


Figure 9: Direct containment of all associations

Finally, association classes are difficult to represent using direct containment. Either a new pointer to the association class instance must be created, or its properties must be captured within the participating classes, violating the domain semantics.

Implementation Constructs

Associations can be implemented using *junction classes* and *container classes* [Fowler97]. Each instance of a junction class has a one directional pointer to each object linked by the association. A container object represents a set of junction object. In Fig. 10, Detection is a junction class and Detection CC is a container class. A Detection object has pointers to a Radar instance and a Flying Entity instance. A Detection CC contains pointers to many Detection instances.

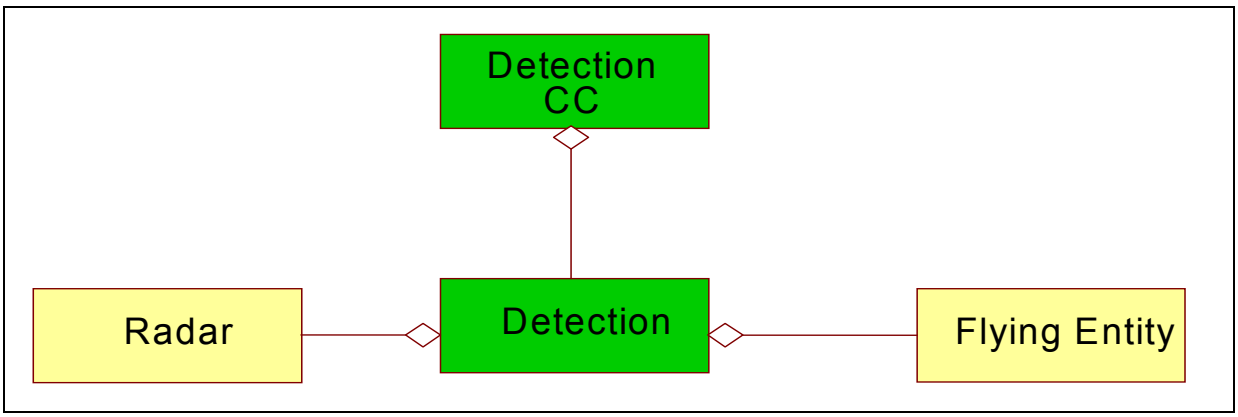


Figure 10 Using constructs to implement an association

Using constructs to implement associations has several advantages. Relationships can be added without changing the domain classes. “This advantage is absolutely critical for large-scale software reuse; otherwise objects need to change every time they are used in a different application" [Whitehurst97]. Furthermore, the container classes can be used for

instance accounting, and designers can apply object-oriented techniques to the container classes themselves.

One or more association classes can easily be added to the association and higher order associations can be modeled by adding more pointers to the junction object. Another advantage is that queries such as "what radars detect any flying entities at time T" can be easily answered. This is particularly important when users follow the progress of the simulation on a GUI. If the associations between the domain objects are buried within the domain objects, the queries must be methods of the domain classes, which pollutes the semantics. For example, a radar that detects a flying entity might not know its identity, so querying on a particular entity may be invalid from a domain perspective.

Another advantage of container class implementation is that queries about associations are easy. Queries such as "what radars detect any flying entities at time T", "what flying entities are detected by radar 124 at time T", and "when was flying entity 88 detected" can be easily answered if a container class of the detection association is used. The answers to such queries are increasingly important when a GUI is used to show simulation activity in wall clock time. If the associations between the domain objects are buried within the domain objects themselves, the queries may become directed and not unidirectional. Additionally, implementing associations this way requires the query mechanism be included with the domain object. Doing so pollutes the semantics of the domain object. For example, does a radar domain object need to support queries such as described above? The actual radar might not know the identification of the flying entity, only that it detects an entity, so querying on an exact flying entity may not be valid from a domain perspective.

4 CONTAINER CLASSES

The domain objects as well as the junction class can be grouped within container classes, leading to higher tractability between the conceptual domain model and the implementation. In Fig. 11 container classes have been added to allow for instance accounting of the domain objects and the association. It is easy to see that the container classes map directly to the conceptual model and that the detection association becomes a first-class entity and is not buried within the implementation.

Container classes provide data structures which can be the fundamental underpinnings of associations implemented as constructs. Using container classes allows the capability to ask questions of the associations (this is called instance accounting).

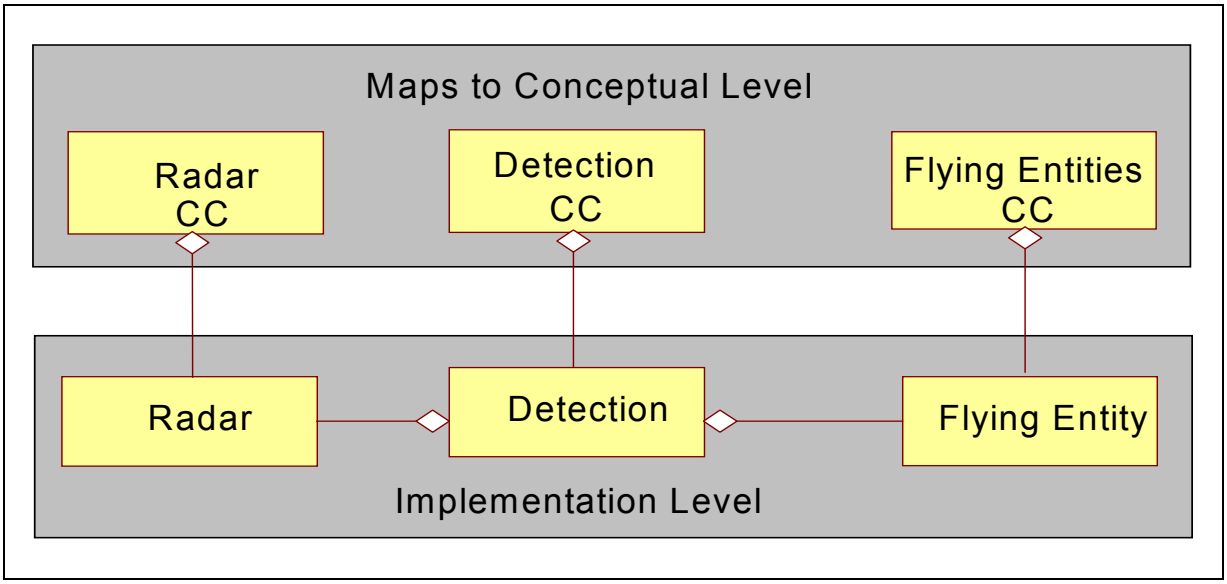


Figure 11: Container Classes

5 CONCLUSION

Associations are critical to the integrity of an object-oriented domain model. It is best to use the constructs of junction classes and container classes to implement associations since they alleviate problems of coupling, cohesion, referential integrity, and semantic misalignment. Using constructs to implementing associations has several distinct advantages. When relationships are implemented as separate classes, more relationships can be added without changing the domain classes. Adding or removing associations will not affect the domain classes. They can be designed without regard to the context in which they are being used, which dramatically increases cohesion and decreases coupling. The container classes used to model the associations can do instance accounting and can be queried so that the relationships can be examined from both sides. Designers can apply object-oriented techniques to the structures that support associations and take advantage of data structures provided by commercial foundation classes.

REFERENCES

[Booch99] Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide Object Technology Series*, ed. Rumbaugh Booch, Jacobson. Reading, MA: Addison-Wesley, 1999.

[Fowler97] Fowler, Martin. *Analysis Patterns: Reusable Object Models*. Menlo Park: Addison-Wesley, 1997.

[Gottlob96] Gottlob, G., Schrefl, M., and Rock, B. "Extending Object-Oriented Systems with Roles." *ACM Transaction on Information Systems 14*, no. 3 (1996): 268-296.

[Kendall99] Kendall, E. "Role Model Designs and Implementations with Aspect-oriented Programming." A paper delivered at the *OOPSLA 1999*, Denver, CO, 10/1999 1999.

[Kristensen94] Kristensen, B. "Complex Associations: Abstractions in Object-Oriented Modeling." A paper delivered at the *OOPSLA 1994*, Portland, OR, 1994.

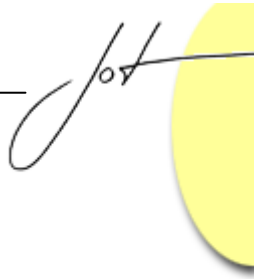
[Odell95] Odell, J., Fowler., M. "From Analysis to Design Using Templates." *Report on Analysis and Design*, March 1995.

[Rumbaugh87] Rumbaugh, J. "Relations as Semantic Constructs in an Object-Oriented Language." A paper delivered at the *OOPSLA 1987*, 1987.

[Rumbaugh91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. *Object-Oriented Modeling and Design*. Englewood Cliffs: Prentice Hall, 1991.

[Rumbaugh99] Rumbaugh, J., Jacobson, I., Booch, G. *The Unified Modeling Language Reference Manual Object Technology*, ed. Jacobson Booch, Rumbaugh. Reading, Mass: Addison Wesley, 1999.

[Whitehurst97] Whitehurst, R. Alan. "Association Frameworks in Simulation Reuse." A paper delivered at the *Computer Simulation Society Proceedings on Object-Oriented Simulation*, 1997.



About the authors

Dr. Charles A. Suscheck is TurboPower's director of research and an innovator of OO technologies. He lectured at OOPSLA and ECOOP, and major companies such as MCI, Raytheon, American Greetings, Sherwin Williams, and Cap Gemini America. Dr. Suscheck has over 20 years of professional experience in IT and holds a Doctorate of Computer Science from Colorado Technical University. His research interests include software development methodologies, discrete event simulations, creative thinking, and the effect of object-oriented frameworks on domain designs. Email: charles@suscheck.com.

Dr. Bo I. Sandén is a Professor of Computer Science at Colorado Technical University. He teaches at the doctoral, masters and undergraduate levels and serves as a doctoral thesis advisor at Colorado Tech. His primary research interest is software design, particularly for multi-threading. He is the author of two textbooks and numerous journal papers. He holds a Ph.D. from the Royal Institute of Technology in Stockholm, Sweden. Before joining academia, he spent 15 years in the software industry as a developer and project manager. He is a member of ACM and of the IEEE Computer Society. Email: reached at bsanden@acm.org.