

# Design Patterns in the Standard Template Library: The Case of Sorted Associative Containers

Arturo J. Sánchez-Ruíz  
CIS Department  
University of North Florida  
4567 Saint Johns Bluff Road, South  
Jacksonville, FL 32224  
USA  
Phone: +1-904-620 2985, Fax: +1-904-620 2988  
arturo@acm.org

## ABSTRACT

We describe the design pattern-oriented and incremental approach we used to extend the Standard Template Library (STL) class of Sorted Associative Containers in the context of a generic programming research problem. This approach allowed us not only to extend the STL by creating yet a new class of sorted associative containers, but also to expose design patterns that were used by STL implementers, even though they did not explicitly followed a design pattern-oriented approach, a practice now customarily supported by several contemporary CASE tools.

## 1. INTRODUCTION

Containers (a.k.a Collections),<sup>1</sup> iterators, and generic algorithms are now common place in mainstream C++ and Java platforms such as the Standard Template Library (STL) [11],<sup>2</sup> Java 2 Standard Edition (J2SE),<sup>3</sup> and the JGL libraries.<sup>4</sup> They are also mainstream concepts in first-year computer science courses, covered by books such as those by Preiss [15], and Ford-Topp [6], to mention just one from the Java and C++ turfs. Moreover, the final draft of ACM/IEEE Computing Curricula lists “Describe how iterators access the elements of a container” as one of the learning objectives associated with the core unit “PL6. Object-Oriented Programming”, part of the Computer Science Body of Knowledge.<sup>5</sup>

<sup>1</sup>In this paper, we use both terms as synonyms.

<sup>2</sup>See also <http://www.sgi.com/tech/stl/index.html>, for the SGI STL Programmer’s Guide.

<sup>3</sup>J2SE v1.4.0 API specification is in <http://java.sun.com/j2se/1.4/docs/api/index.html>.

<sup>4</sup>For more information, see <http://www.recursionsw.com/products/jgl/>.

<sup>5</sup>The entry point to the document is <http://www.computer.org/education/cc2001/final/index.htm>. A direct link

Simply put, generic algorithms are expected to operate on a wide variety of containers (which are just that, holders of generic objects), which implies that they cannot resort on knowledge associated with underlying data structures and the type of constituent elements. To achieve such orthogonality, iterators come to the rescue, acting as mediators between algorithms and containers. When the former need to get a hold of a particular element in a container, they use the latter as proxies.

A typical iteration programming pattern these algorithms utilize, usually referred to as a traversal, is that of going through all the elements in the container exactly once. When containers allow for repetition of equivalent elements (e.g. multisets, and multimaps in STL and JGL), the traversal should go through all equivalent elements exactly once.

Let  $C$  be a container and  $F(x, C)$  a function, and consider the family of traversals such that given an initial value of  $C$ ,  $F$  is invoked after each element is fetched from it. We have studied the case for which  $F$  can potentially insert new elements into  $C$  while the traversal is in progress.

This problem can be considered as trivial for linear containers, such as vectors, dequeues, queues, and lists, simply because one can arguably force insertions to be performed at a prescribed end of the container in question. This is not the case with Sorted Associative Containers (SAC), which use a non-linear ordered data structure (typically some kind of balanced tree) to achieve efficiency, and therefore programmers cannot have control on where new elements are inserted. The moral is therefore that this iteration pattern could yield unpredictable results in general, which is the reason why none of the mentioned collection frameworks supports it.

We have formally characterized this iteration pattern using standard rewriting systems theory, and proven sufficient conditions for its finiteness and determinacy, i.e. conditions under which it terminates with the same unique container, regardless of the order used to traverse the elements in the container supplied as input. We call these iteration patterns

to the CS-BOK is <http://www.computer.org/education/cc2001/final/chapter05.htm>.

complete traversals [12].

In section 2 we discuss the problem of extending the STL to support complete traversals using design patterns in an incremental fashion. This approach proved to be useful for better understanding and exposing design decisions made by STL implementors, expressed as patterns, even though they did not originally followed this approach. Such knowledge made the exercise of extending the STL easier.

The last two sections wrap up the paper, with a discussion on other approaches that have appeared in the literature in connection with the issues presented in this paper, followed by our conclusions and plans for future work, respectively.

## 2. EXTENDING STL SORTED ASSOCIATIVE CONTAINERS

Suppose that a given container-function pair  $(C, F)$  defines a complete traversal, which means that any traversal pattern that goes through all elements in  $C$ , exactly once, calling  $F$  at each iteration, which can potentially add new elements to  $C$ , always terminates with the same final container, regardless of the order in which elements are retrieved. This implies, in particular, that an iteration pattern like the one shown in figure 1 should always terminate with the same container.

```

Container C;
Functor F;
C::iterator it;
for (it = C.begin(); it != C.end(); ++i)
    F(&C, *i);

```

Figure 1: A naive iteration would not handle complete traversals.

It is easy to see that such a naive approach would not work in general. Consider, for instance, the case for which the element just inserted becomes the predecessor (according to the traversal order) of the current element. The goal is therefore to extend STL Sorted Associative Containers (SAC) so that they seamlessly and transparently support complete traversals. In this section we shall show how we incrementally used design patterns to accomplish this goal.

The first step in our incremental approach consists of expressing the relationships among containers, generic algorithms, and iterators using GOF patterns. We use these patterns as a guide to read the Silicon Graphics Inc. (SGI) implementation of STL<sup>6</sup>, which is in turn based on the original Hewlett-Packard (HP) implementation by Stepanov, Lee, and Musser. This reading allows us to enrich the description in terms of GOF patterns. The process is continued until all elements that are needed for implementing the desired extension are unveiled, time at which implementation details in connection with such extension can be presented.

<sup>6</sup>Available for downloading from <http://www.sgi.com/tech/stl/download.html>.

## 2.1 GOF Patterns and STL SACs

GOF Iterator pattern [8] captures the fundamental relationships among containers, iterators, and generic algorithms, in a language-independent way (c.f. figure 2). Namely, generic algorithms play the role of clients which get to container elements by using method `createIterator()`, and those associated with class `AbstractIterator`, appropriately implemented by the corresponding `ConcreteIterator` class, using the representational knowledge of class `ConcreteContainer`.<sup>7</sup>

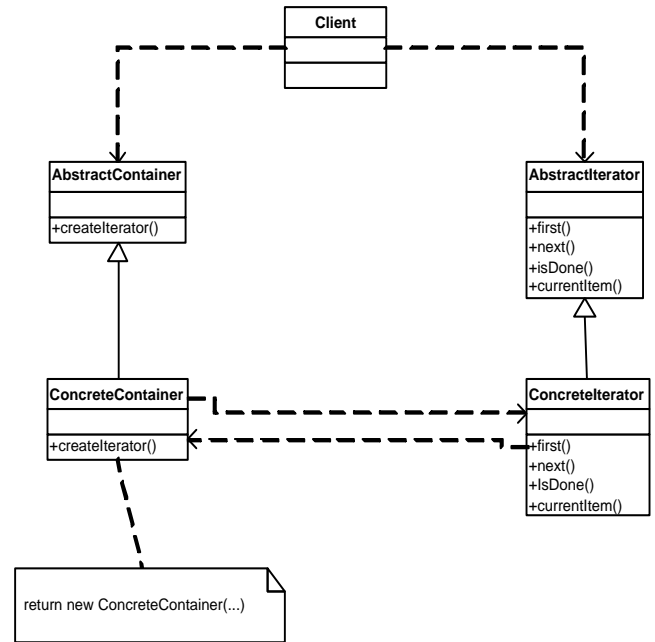


Figure 2: GOF Iterator Pattern.

How is this pattern rendered by the STL in C++? After examining the implementation associated with generic sets,<sup>8</sup> one finds:

- (a) Generality with respect to the type of constituent elements is achieved by the use of templates, not inheritance.
- (b) Another class encapsulates implementation details associated with a data structure that provides for fast implementation of core functionality such as insertions, retrievals, and queries. This is also a template class, which at first sight looks quite complicated in the context of a set. We shall see later where this complexity pays off. The data structure in question is a red/black tree [4].
- (c) Typical methods associated with iterator objects are implemented as overloaded operators by a template class which knows all required details in connection with the tree-based data structure.

<sup>7</sup>`AbstractContainer` and `AbstractIterator` could be abstract classes or interfaces.

<sup>8</sup>The involved files from the SGI implementation are `stl_set.h`, and `stl_tree.h`.

Figure 3 shows how the design pattern is rendered for the STL implementation of sets. Notice that not only the iterator pattern is used, but also the (object) adapter, where `Set` is playing the role of adapter, and `Tree` is playing the role of the adaptee.

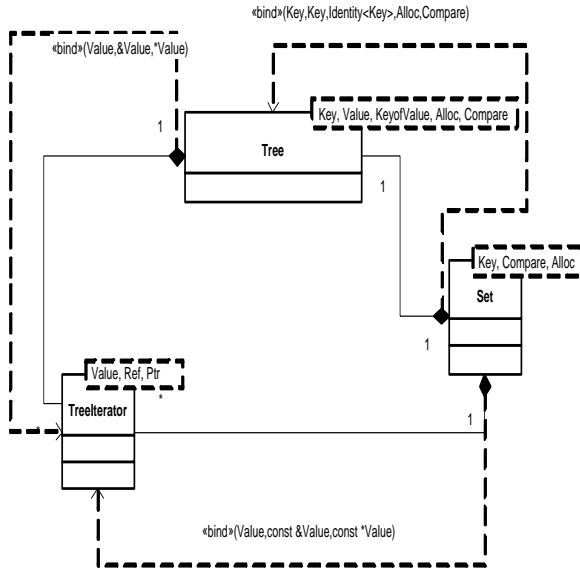


Figure 3: STL Set design rendering.

There is yet another pattern that is being used in a subtle way, which is the key for solving the following practical problem. Figure 4 shows the taxonomy of STL SACs. Associativity refers to the fact that elements in the container can be pairs (*key, value*), or just plain *values*. Repetition refers to whether equivalent elements are allowed.<sup>9</sup> In summary, `Sets` are collections of *values* for which repetition is not allowed. `Maps` are collections of (*key, value*) pairs for which repetition of *keys* is not allowed. `MultiSets` and `MultiMaps` are the dual counterparts for which repetition is allowed. The problem is, how to implement containers that can hold both non-associative and associative elements, with the possibility of allowing for repetition of equivalent elements, by using *the same* backing data structure?

Type of SAC	Associative?	Repetition?
Set	No	No
Multiset	No	Yes
Map	Yes	No
Multimap	Yes	Yes

Figure 4: Taxonomy of STL SACs.

We are now ready to explain the meaning of template parameters associated with class `Tree`.

- **Key**: is the class associated with keys in the container.
- **Value**: is the class associated with values in the container, when they are associative.

<sup>9</sup> A strict weak order  $<$  is supposed to be defined on the collection elements, which implies that  $x$  and  $y$  are equivalent if and only if both  $\neg(x < y)$  and  $\neg(y < x)$  hold [11].

- **KeyofValue**: is the class associated with functors<sup>10</sup> which are used to automatically decide whether constituent elements are pairs (associative containers) or not.
- **Alloc**: this is the class associated with allocator objects which hide details in connection with memory allocation. We will not discuss allocators in this paper. For more details, see [11, 14].
- **Compare**: the class of functors that define the strict weak order required by the tree structure.

In order to show how the design in figure 3 is repeated for multisets, maps, and multimaps, let us consider the way each of these instantiate template class `Tree`, using their own template parameters.

- `MultiSet<Key, Compare, Alloc>` instantiates `Tree` as `Tree<Key, Key, Identity<Key>, Alloc, Compare>`. Just like `Sets`. Notice that the non-associativity is expressed by repeating the second argument, and by using the generic functor class `Identity` (in file `stl_function.h`), which produces objects that are nothing more than identity functions.
- `Map<Key, T, Compare, Alloc>` instantiates `Tree` as `Tree<Key, pair<const Key, T>, Select1st<pair<const Key, T>>, Alloc, Compare>`. Where `T` is the associative component, and `pair` is a generic class that builds pairs of objects of the given types. Template class `Select1st` (in file `stl_function.h`) construct functors that select the first component of the pair provided as argument. This functor is then used when comparing values using the ordering functor from class `Compare` and therefore only keys are compared.
- `MultiMap<Key, T, Compare, Alloc>` instantiates `Tree` just like `Maps`.

This solves the problem as far as using the same data structure for associative and non-associative containers.

How is the uniqueness/repetition issue handled? Simply by defining two different methods for inserting elements into the data structure (`insert_unique`, and `insert_equal` for the case of red/black trees, in file `stl_tree.h`). Since the SACs act as the adapter of class `Tree`, each kind uses the method that fits its needs.

We are now ready to show the design pattern that captures STL SACs, depicted in figure 5. In our notation `PARS` denotes the sequence of template parameters that are required by each SAC class, and the binding just signifies that such substitution is a function of `PARS` as indicated above. Notice that this time, besides the adapter and iterator patterns, an instance of the command pattern is also present, because of the roles being played by all required functors [1].

<sup>10</sup> A functor is just an object of a class that overloads `operator()`. As a result, objects from such classes can be thought of as functions with a state.

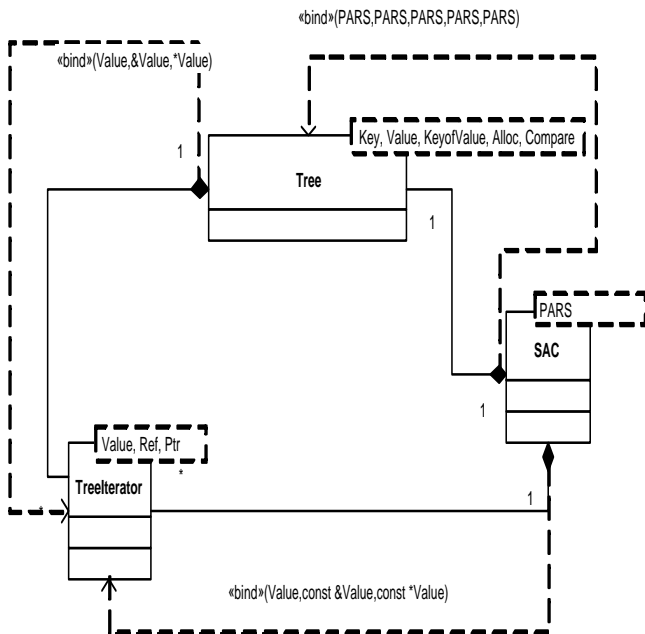


Figure 5: STL SAC Design Pattern.

A *complete container* is one that seamlessly and transparently supports complete traversals. The following sections show three approaches to enriching the STL with complete containers, using design patterns.

## 2.2 Complete Containers Using the Adapter and Command Patterns

Figure 6 shows the application of the adapter pattern to create class CCA, which adapts a given SAC. Briefly, the constructor for the new class receives a reference to the SAC, which is used to create a mirror image of it on a list (of references to the elements in the container). Insertion methods are overloaded in such a way that elements are inserted in the SAC preserving the underlying order, and *appended* to the list. Finally, corresponding iterator objects are simply iterators on the corresponding list. This class therefore keeps two orthogonal views of the same container, and as a result is able to support complete traversals. We have not added details associated with template parameters for the sake of brevity and clarity.

The second approach, shown in figure 7, uses generic functions and the command pattern, by means of functors. Notice that we are using an ad-hoc notation to represent generic algorithms for the lack of one in UML [16]. In this case, the traversal associated with pair  $(C, F)$  is implemented through standard container iterators, but it is now functor  $F$  who keeps a queue with all new elements that are generated after each iteration. Using this queue, generic functions can keep track of the order in which elements should be processed so none is lost. We have implemented generic functions that deal with unique and non-unique containers.

More details on the C++ implementation of these components, as well as a complexity analysis is presented else-

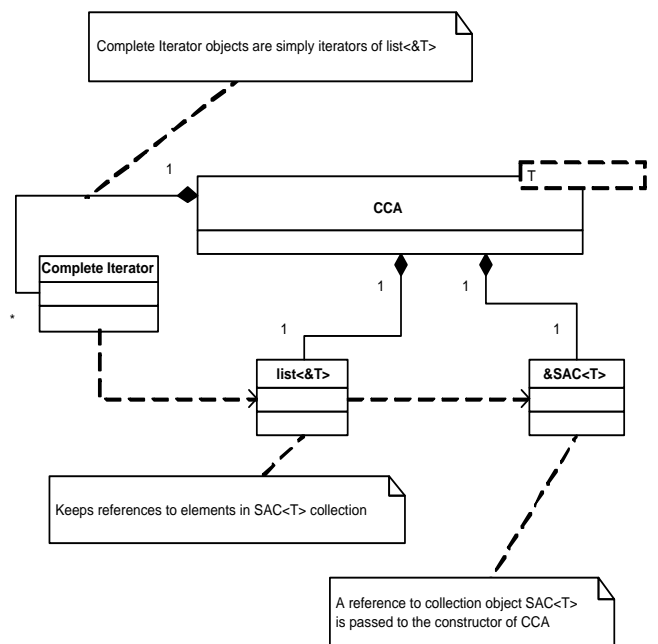


Figure 6: Complete Containers using the Adapter Pattern.

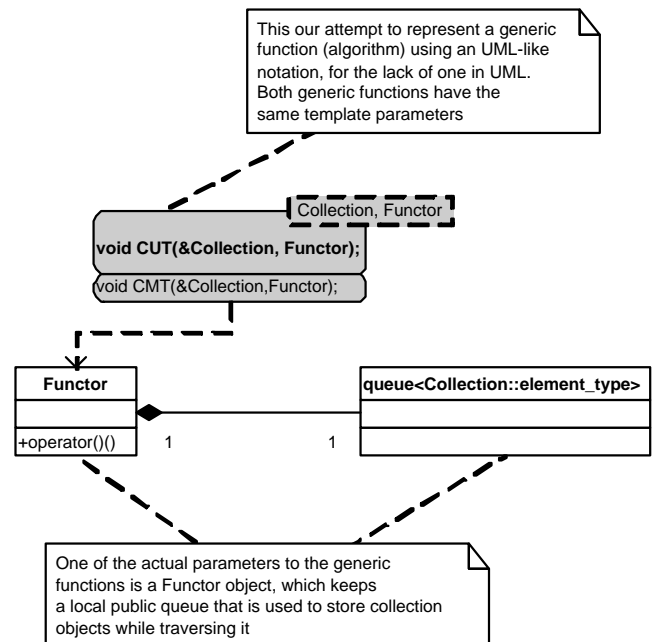


Figure 7: Complete Containers using generic functions and the Command Pattern.

where [7]. The code is also available.<sup>11</sup>

### 2.3 Mimicking the SAC Pattern to Extend the STL

In order to explore approaches that did not use so much extra storage, we mimicked the design pattern behind SACs in such a way that the core data structure that supports them was able to inherently handle complete traversals using specialized iterators. The resulting pattern is presented in figure 8. We have avoided details associated with template parameters and their bindings for the sake of brevity and clarity.

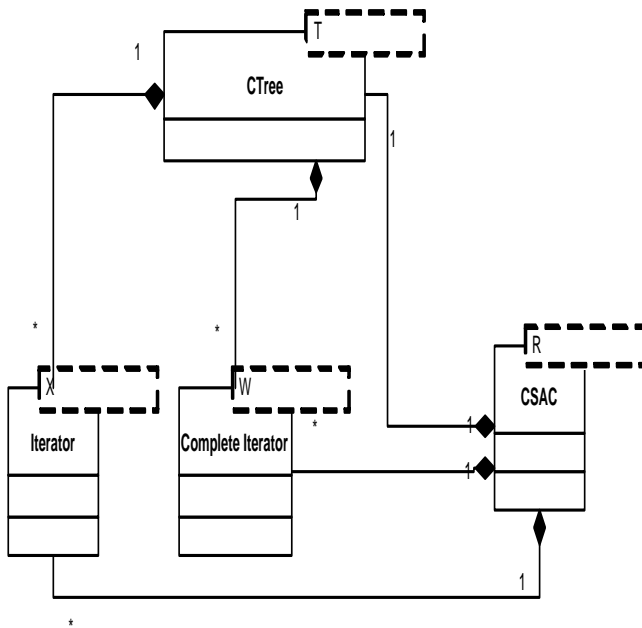


Figure 8: Using SACs Design Pattern to create CSACs.

The idea is based on modifying the node structure associated with the red/black trees in such a way that they are kept (bidirectionally) threaded in the order of insertion. The new class of iterators (complete iterators) bidirectionally traverse the tree structure using these new links, which are used to handle complete traversals. CSACs also support all iterators SACs do. The resulting code is available.<sup>12</sup>

### 3. RELATED WORK

Eichelberger and Gudengberd [5] present a partial UML characterization of STL but do not identify design patterns. We are not aware of any previous work analyzing STL in light of the concept of design patterns.

The so-called class adapter pattern [8] (i.e. the adapter inherits from the adaptee), is used by the implementation discussed in Plauger, Stepanov, Lee, and Musser [14]. A known weakness of this flavor, with respect to the object adapter, is that it cannot handle the adaptation of potential subclasses

of the adaptee. The authors, however, do not discuss their implementation in terms of design patterns.

Although various books in the literature associated with the teaching of Data Structures use STL as a design exemplar [10, 3, 2, 6], they end up following the traditional approach which focuses on implement aspects of algorithms that are needed to manipulate structures such as trees and linked lists. We argue that such an approach misses the fundamental point of showing examples of good design (and how these are derived), an issue that is put in the background and obscured by flat C++ code, which could be perceived as complex by students.

We discuss the performance assessment of all generic components discussed in this paper, and present some related theoretical results elsewhere [9].

### 4. CONCLUSIONS AND FUTURE WORK

The case presented in this paper can be used as a constructionist example<sup>13</sup> to show students a practical application of design patterns, and also to reinforce the fact that elements in the GOF catalogue are indeed present in many exemplary designs.

We are currently working on developing a pattern-oriented approach to teaching introductory computer science courses, which relies on deriving patterns and showing their realizations in exemplary frameworks such as the STL, JGL, J2SE, and J2EE using CASE tools that support patterns in an extensible way such as Together Control Center<sup>14</sup>.

### 5. REFERENCES

- [1] A. Alexandrescu. *Modern C++ Design. Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
- [2] T. Budd. *Data Structures in C++ Using the Standard Template Library*. Addison-Wesley, 1997.
- [3] W. J. Collins. *Data Structures and the Standard Template Library*. McGraw-Hill, 2003.
- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
- [5] H. Eichelberger and J. W. v. Gudenberg. UML description of the STL. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 2000.
- [6] W. Ford and W. R. Topp. *Data Structures with C++ Using STL*. Prentice-Hall, second edition, 2001.
- [7] E. Gamess, D. R. Musser, and A. J. Sánchez-Ruiz. Complete traversals and their implementation using the standard template library. *CLEI Electronic Journal*, 1(2), 1998. Available from <http://www.dcc.uchile.cl/~rbaeza/clei/>.

<sup>11</sup><http://www.cs.rpi.edu/~musser/gp/traversals/>

<sup>12</sup><http://www.unf.edu/~asanchez/CSAC/>

<sup>13</sup>In the sense that is based on Papert's *constructionism* [13].

<sup>14</sup>See <http://www.togethersoft.com>

- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns CD*. Addison-Wesley, 1995.
- [9] W. Klostermeyer, D. R. Musser, and A. J. Sánchez-Ruiz. Complete traversals as general iteration patterns. In J. Gibbons and J. Jeuring, editors, *Working Conference on Generic Programming 2002*. Kluwer, 2002.
- [10] W. H. Murray and C. H. Pappas. *Data Structures with STL*. Prentice-Hall, 2001.
- [11] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, second edition, 2001.
- [12] D. R. Musser and A. J. Sánchez-Ruiz. Theory and generality of complete traversals. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 91–101. Springer-Verlag, 2000.
- [13] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, 1980.
- [14] P. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser. *The C++ Standard Template Library*. Prentice-Hall, 2001.
- [15] B. R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley & Sons, 1999.
- [16] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.