

UML Description of the STL

H. Eichelberger, J. Wolff v. Gudenberg
Universität Würzburg
{eichelberger,wolff}@informatik.uni-wuerzburg.de

Abstract. In this paper we show how the specification of the Standard Template Library STL and its implementation can be described by UML diagrams. We define appropriate stereotypes to describe STL concepts like containers, iterators, function objects and global algorithms. For the graphical description of the implementation of the STL we extend the UML metamodel.

1 Concept, Category

The Standard Template Library STL is a collection of containers and algorithms, combined by explicit iterators and parameterized by function objects (functors). Specific adapters or allocators can be used. All these key items are described by means of a concept [1] which is called a category in [2] in a slightly more general sense. The term theory is used in [4]. A concept may be regarded as a set of requirements that are fulfilled by each model. An equivalent characterization is a set of types as models. Thirdly, a category can be viewed as a set of programs which use models of it to exploit the requirements. Theoretically a concept may be considered as a set of multisorted algebras.

Each concept acts as a parameterized metatype whose models are the templates or types which make up the STL. It usually contains a set of types and operators specifying the syntactically correct application of models (i.e. instances). Furthermore, semantic information, such as invariants or requirements on algorithmic complexity, is given.

Concepts can be refined. A refined concept fulfills all requirements of its base concept, i.e. all types are defined and all expressions built with the operators remain valid. Refinement is a reflexive, transitive relation, very similar to generalization or inheritance in the sense that a refinement owns more features or fulfills more requirements. A model of a refinement of a concept C hence is a model of C .

2 UML

2.1 Different layers of abstraction

The Unified Modeling Language UML is a graphical modeling language providing various diagrams to illustrate all phases of a software development process [6]. Class diagrams describe the static structure of the software on the class level. They contain classes and their associations, generalizations and other dependencies. A class usually contains attributes and operations but more compartments may be added.

On the object level the individual objects are drawn in object diagrams where objects as instances of specific classes are connected by links as instances of associations. A class hierarchy is not shown on this layer. Object diagrams and class diagrams hence concern different levels of abstraction. In the UML this layered approach is carried out to a further level of abstraction, the metamodel layer or model element level that shows the relations of the model elements like “class” or “association”. Class diagrams are used also on this level. In the current version of the UML no “vertical” relations between different layers of abstraction are drawn explicitly, the class name is part of the object name, instead.

Stereotypes are used in the UML to group model elements, they prescribe the desired usage or variant. A new graphical notation can be introduced as well. Stereotypes are one of the incorporated extension mechanisms of the UML. It is, e.g., possible to define a stereotyped class `STLClass`, that contains an additional type compartment. Extension of the UML by definition of new stereotypes, constraints, and tagged values is called a UML profile. The extension of the metamodel itself that is necessary for the description of the STL implementation is formally given in [3] where the semantics of the newly introduced stereotypes may also be found.

2.2 Templates in UML

UML allows the parameterization of each model element with other model elements. Those templates, however, are not considered as first-class citizens of a model. Class templates, e.g. may not be inherited nor serve as sink of an association. Template parameters are names or placeholders. No semantics or relations to other diagram elements can be specified. The instantiation of (class) templates is illustrated by a `<<bind>>` dependency where the template arguments, i.e. the classes which instantiate the template parameters, are listed or by an extended class name in C++syntax.

3 UML description of STL

We use class diagrams on the class level and on the metamodel level to illustrate the functional specification of the STL. We have made the experience that a

graphical representation helps in teaching, clarifying, and comprehending the structure of the STL. Each kind of classifier, i.e. a class-like structure, is marked with an appropriate stereotype to emphasize the particular purpose.

3.1 Functional Specification

A category is a new model element that is depicted as a rectangle consisting out of up to three compartments. The first mandatory compartment contains the name in bold face marked by the stereotype indicator `«Category»`. The second compartment lists the contained types, the third the available operators or operations. Both are optional. A category is a kind of a powertype, i.e. a classifier whose instances in general are templates for types or classes. Actually, most categories are themselves parameterized with the types in their type compartment. A type compartment consists out of a mapping from external type names to internal names (cf. typedefs in C++).

Notation: `internal : external = default`

If the external names are unbound, they denote template parameters of the category. The default instantiation may be given. To keep the diagrams simple we do not explicitly show template parameters of categories.

A category is a generalizable element, that means that it can be specialized and the specialization, the inheriting category, contains all types and operations of its ancestor. The functional specification of iterators, containers etc. can be given as a hierarchy of categories.

Although there are a lot of different UML model elements to describe behavior, there is no such element precisely modeling global functions in C++. We model a global function (an algorithm) as a classifier (`«Function»`) containing its set of parameters and result type. Note the difference between a category describing a functor on the model element level and the description of the interface of an algorithm by `«Function»` on the class level. This stereotype indicates that the algorithm does not belong to any class.

3.2 Implementation

For the presentation of the implementation of the STL we have to extend the UML even more.

We introduce a new dependency `«modelOf»` between an implementing class (template) and a category. This relation leads from the class level of abstraction to the model element level. We emphasize this fact by shading categories in a class diagram. Note, that a class can be model of several categories. The `«modelOf»` dependency propagates to each instantiation of a template and by UML semantics to each descendant of a base class.

Requirements on argument or result types of algorithms can be graphically illustrated by showing the `«modelOf»` dependency. For the description of template parameters we introduce the stereotype `«Template Parameter»`. This explicit

notation of the template parameters allows to depict relations between different parameters as well as the enumeration of all concepts being modeled. One such dependency between types is the `«convertsTo»` relation which indicates that the source type is convertible to the destination type. In UML there is a `«bind»` dependency from an instantiating class to a template. In our profile a `«boundBy»` dependency leads from the template symbol to each of its parameter descriptions. To make that relation more obvious we extend the template notation of the UML where template parameter names are contained in a dashed rectangle in the upper right corner in the way that individual parameter rectangles are possible and that those rectangles can be source of a `«boundBy»` dependency.

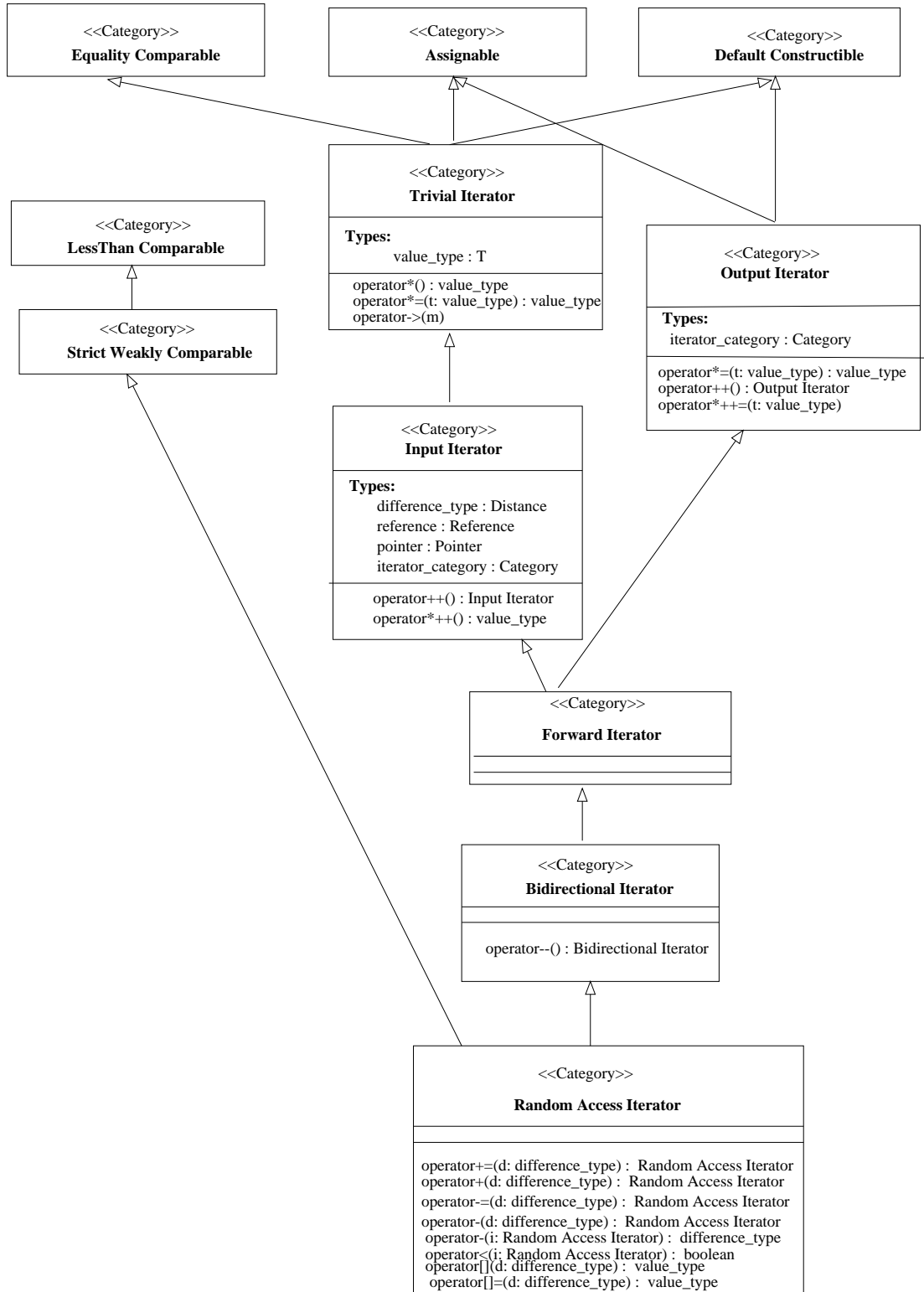
A formal description of these extensions is given in [3].

4 Sample Diagrams

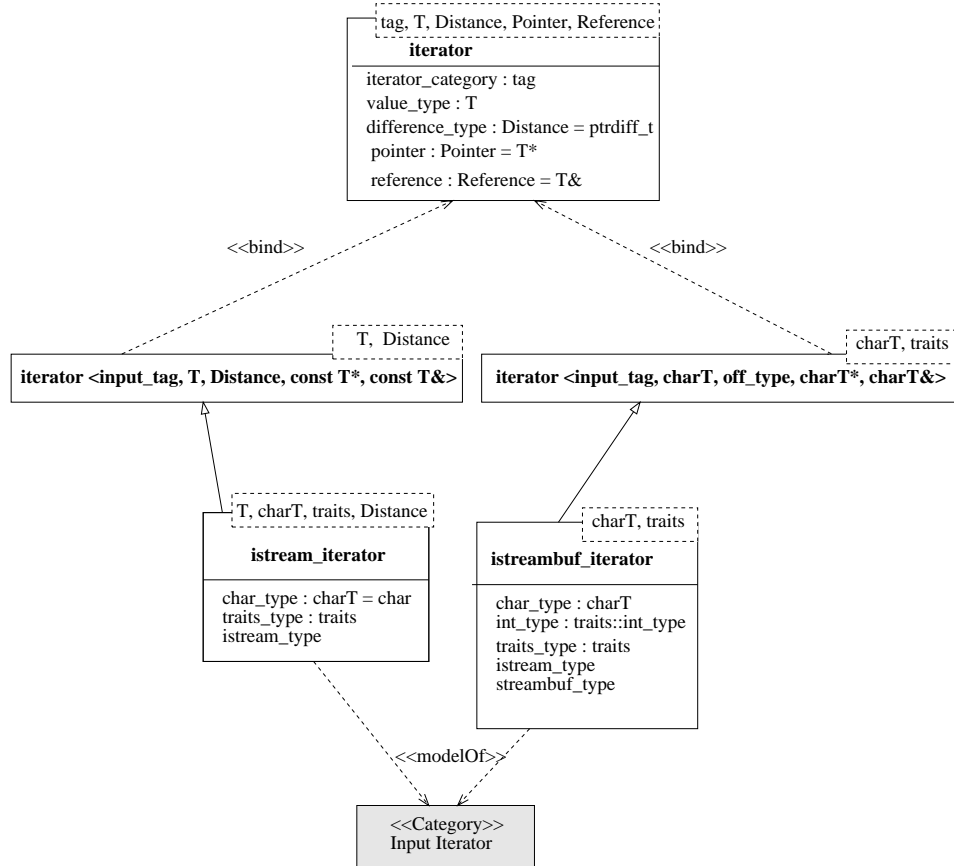
In this section we show some sample diagrams. A comprehensive description of the STL is given in [3].

4.1 Iterators

The following diagram shows the functional specification of the iterator categories as a hierarchy.



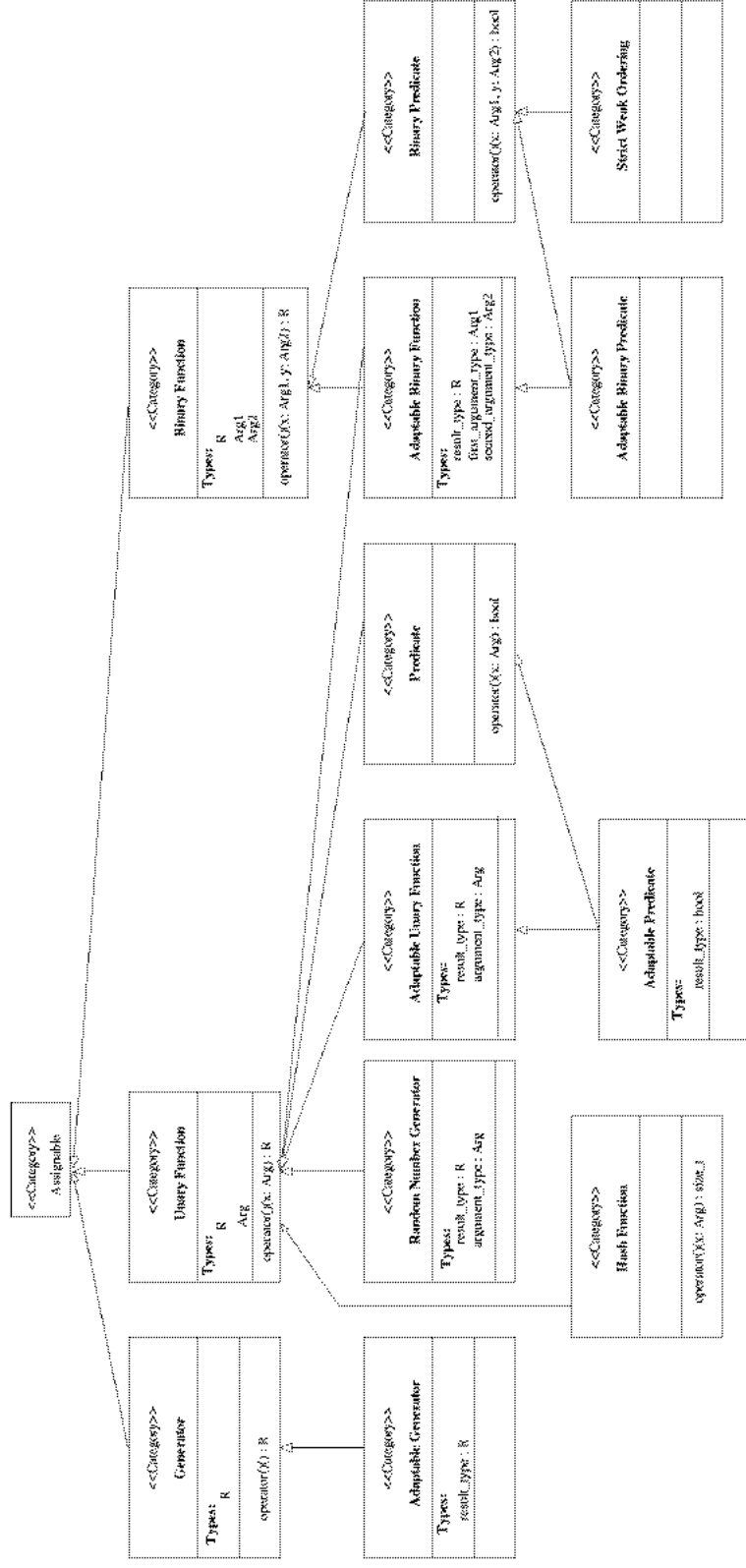
The following models of input iterators are contained in the STL.



The tag type is an enumeration of the different iterator categories which enables the definition of the common base class `iterator`. The detailed mechanism of instantiating the `iterator_traits` template is not reflected in this diagram. More iterators are provided by the containers.

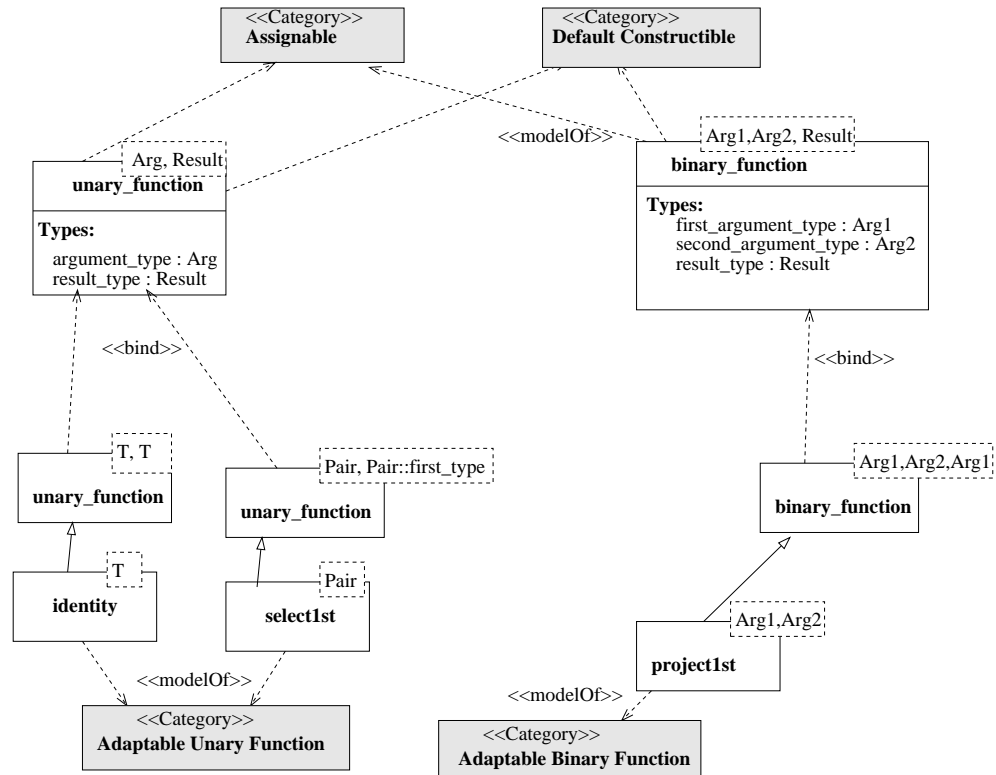
4.2 Functors

Functors, i.e. categories for templates or classes that encapsulate function objects are grouped by the number of parameters.



In Generator, Unary Function, and Binary Function no internal type names are specified.

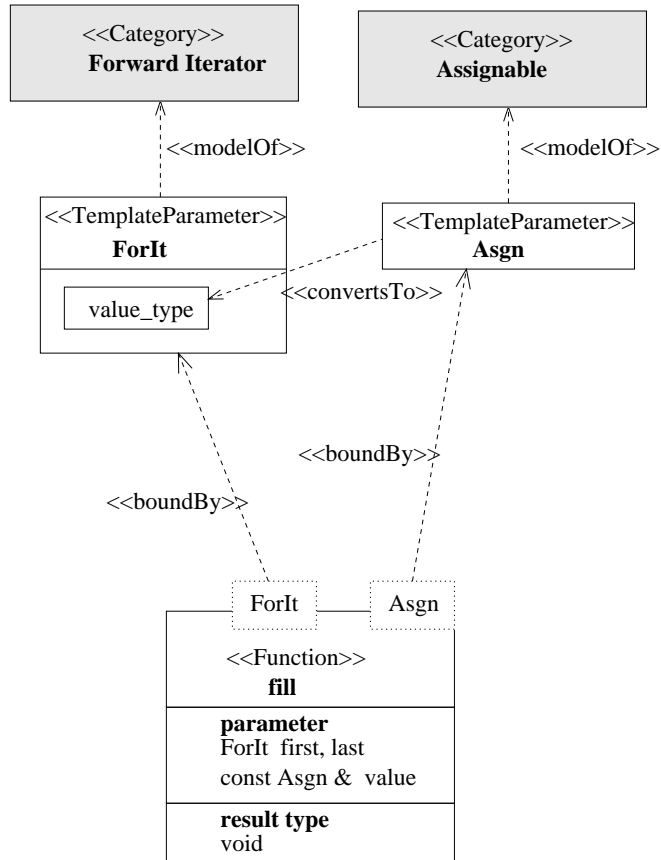
Instances of these functors are provided for different application purposes. We show a projection and a selection function object. Note that the operation is always obtained by overloading the () operator.



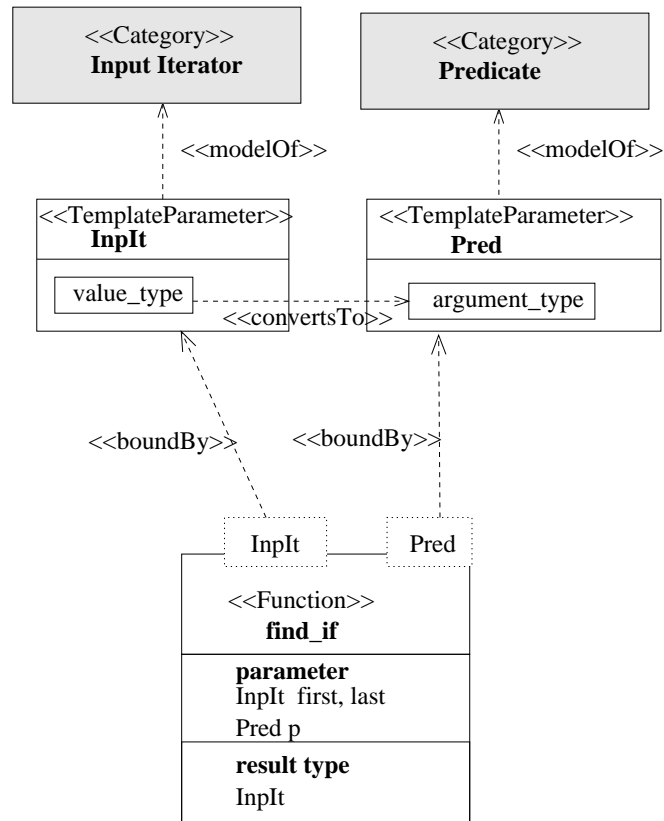
This diagram has to be read as follows: `unary_function` is a template which defines typenames for its two template parameters. `identity` inherits from the partial specialization `unary_function<T, T>`, it is a model of `Adaptable Unary Function`, and hence overloads the () operator...

4.3 Algorithms

The following two diagrams illustrate the specification of algorithms.



`fill` is a function template whose first template parameter is a model of the category Forward Iterator. The second template parameter may be any assignable type that is convertible to the forward iterator's value type. The next diagram is to be interpreted analogously.



5 Summary

We have shown that a layered approach and some simple UML extensions open a way to graphically specify the functionality of the STL and illustrates its implementation. Further work will be done to develop a visual representation of generic programs in general.

References

- [1] M. Austern: *Generic Programming and the STL*, Addison-Wesley, 1998
- [2] J. Barton and L. Nackman: *Scientific and Engineering C++*, Addison-Wesley, 1994
- [3] H. Eichelberger, J. Wolff v. Gudenberg: *A UML Profile for the Description of the STL*, Tech. Report, Institut für Informatik, Universität Würzburg, 2000, to appear.

- [4] J. Goguen: *Parameterized Programming*, IEEE Transactions on Software Engineering, Vol SE-10,5, 1984, pp.528-543
- [5] J. Rumbaugh, G. Booch, I. Jacobson: *Unified Modeling Language Reference Manual* Addison-Wesley Longman, 1999
- [6] *OMG Unified Modeling Language Specification* Version 1.3, June 1999 via <http://www.rational.com>