GENERATING FLEXIBLE CODE FOR ASSOCIATIONS

Mayer Goldberg and Guy Wiener
Department of Computer Sciences, Ben-Gurion University, 1 Ben-Gurion Blvd, Be'er-Sheva, Israel

Keywords: Associations, Code generation, Model-driven Architecture, Meta-programming.

Abstract: Generating code for associations is one of the most fundamental requirements from a model-based code generator. There are several approaches for implementing associations, ranging from using basic collections frameworks to using a database. The choice between them derive from the current requirements of the software: Whether parallelism, caching or persistency required for a relation. Hard-coding a specific design choice makes it difficult to alter it later. In this work, we propose a scheme that allows for automatic code generations of associations with different features, without requiring manual changes to the code. These features include using indices, traversing the association in parallel, or using an external database. Instead of the sequential iterator interface, we propose to use an interface that is based on operations over collections, such as *Foreach*, *Filter*, *Map* and *Fold*. This interface allows for writing operations that traverse the association without being dependent on the implementation details of the generated code.

1 INTRODUCTION

Generating code from UML class diagrams requires translating the high-level UML specifications into concrete statements in some programming language. UML specifications differ from statements in Object-Oriented Programming Languages (OOPL) in the following aspects:

1. UML statements have a richer semantics then their OOPL counterparts: Not every element in a UML diagram can be expressed as a single statement in an OOPL.
2. UML is more abstract then OOP code: It specifies the expected structure and behavior of software components, not how to implement it.

The gap between the semantics of UML associations and OOPL has been discussed extensively in the UML literature. The full semantics of UML associations, as described in UML reference manual (Rumbaugh et al., 2004), includes bi-directionality and multiplicity constraints, as well as more advanced features that have no direct equivalence in OOPLs: Relations between associations (such as subset or redefinition), association classes, and general OCL constraints[1] . In their seminal work on Object-Oriented Analysis and Design (OOAD), Martin and Odell discuss this problem (Martin and Odell, 1992). They suggest to implement bi-directional associations by using either pairs of references or association objects. Craig Larman suggests refining bi-directional associations to uni-directional ones (Larman, 2004). Several works (Fowler, 2003; Gessenharter, 2008; Akehurst et al., 2007) offer different implementations to associations that preserve more of their original semantics. It is clear from these works that implementing associations requires several code statements. This fact, as well as the effort involved in changing the implementation, suggests that code generation would be useful.

The second item, namely that UML is more abstract then OOP code, is commonly overlooked in the literature. The gap between the specification and the code is intentional. Harrison et al., in their detailed work on mapping UML specifications to Java, explain that a model expressed independently of a specific implementation provides greater flexibility (Harrison et al., 2000). This flexibility is required: According to (Pigoski, 1996), the cost of maintenance is 90% from its development cost, and (Martin and McClure, 1983) show that 80% of the maintenance cost are dedicated to *perfective activities*[2] and *adaptive ac-*

[1]For details on OCL, see (OMG, 2005; Warmer and Kleppe, 2003)

[2]Activities that improve the software

Goldberg M. and Wiener G. (2010).
GENERATING FLEXIBLE CODE FOR ASSOCIATIONS.
In *Proceedings of the Fifth International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 96-104
DOI: 10.5220/0002999700960104

tivities[3]. Therefore, any re-use of the *model* that is independent of a platform or performance constraints can lower the cost of maintenance. Many UML statements can be implemented in more than one way.

The semantics of UML associations only specifies the relations that the system should maintain. The details of how to access, traverse, and modify the relation are left for the implementer. All of the works mentioned above focus on providing a single implementation to associations, based on standard collections frameworks — For example, see the Java collections tutorial (Bloch, 1995). This approach provides the richer semantics of associations in code, but does not allow for replacing the implementation without re-writing the code. For example, most standard collections frameworks provide only sequential traversal over an entire collection. This limitation neither appears nor is implied by the UML standard[4]. We would like to allow the developer to choose between sequential and parallel traversal by changing a property of the association, without re-writing the code. Fast access to elements in a collection with specific values requires in most frameworks adding a supporting data structure explicitly (E.g., a hash-table that maps from integers to persons in a collection that are of a given age). We would like to allow the developers to add such indices as a property of the association, or even let a smart code-generator decides which indices should be used.

In this work we present flexible code generation scheme that allows for replacing the generated implementation of associations without requiring changes to the rest of the code. Section 2 describes the interfaces for accessing, updating, and traversing over associations. Section 3 describes different schemes for generating implementations for those interfaces. Section 4 provides an example of implementing and using this scheme to improve the implementation of associated classes. Section 5 concludes.

2 INTERFACES

To replace the implementation of associations independently of the client code[5], we must provide uniform interfaces for operations over associations. These operations include:

1. Adding and removing pairs of associated instances from the relation.
2. Traversing over the instances associated with a given object.

We ignore the case of changing the set of instances that are associated with a given object, since it can be implemented by adding and removing pairs from the relation. Operating on pairs instead of unrelated collections allows the implementation to enforce bidirectionality and multiplicity constraints. This approach is common to many of the works mentioned above.

The second kind of operations, traversing associated instances, requires special attention. Most work on associations propose to use a variation of the Iterator pattern (Gamma et al., 1995). This pattern provides a way to perform a sequential traversal over a data structure without exposing its implementation. This approach, however, has several drawbacks:

1. It forces the traversal to be sequential and does not enable parallel traversal, even when there are no data dependencies between the iterations.
2. It does not encapsulate the selection of elements. Therefore, optimizations like using hash-tables or caching must be a part of the client code.
3. Iterators, together with loop commands (`for`, `while`, etc.) often serve for implementing operations over a range of elements. The body of the loop represents an operation over a single element and is dependent on its internal structure. The iterator pattern encapsulates the traversal, but exposes the structure of the traversed elements. Moving this implicit operation to a method in the class of the element and explicitly applying this method on all associated instances would provide better encapsulation.

To overcome these drawbacks, we propose to use different set of interfaces, similar to the one outlined in (Martin and Odell, 1992):

- The *Association* interface represents the relation between two types.
- The *AssocEnd* interface represents the collection of instances that are associated with a given object.
- The *Foreach*, *Map*, *Filter*, and *Fold* interfaces represent operations over a collection of instances. They are called *aggregator interfaces*.

The aggregator interfaces are *generated* and include sub-sets of the messages from the associated

[3] Activities that adapt the software to new platforms

[4] Class diagram specifications do not discuss any operations over associations. The relevant sequence diagrams specifications, such as for loops and multiple instances, also omit these details. See the UML reference manual (Rumbaugh et al., 2004).

[5] "Client code" here refers to the code that uses the associations, as opposed to the code that implements the associations

type. The implementations of these interfaces are also generated and include the concrete operations over the collection. The *Association* and *AssocEnd* interfaces are not generated, since that their operations signature are constant. However, the implementations of *AssocEnd* are parametrized by the concrete type of aggregators that they returns. Therefore the implementations of *AssocEnd* are either generated, or take *AssocEnd* factories as parameters (See the *factory* design pattern at (Gamma et al., 1995)). Similarly, the implementation of *Association* is also parametrized by the concrete type of *AssocEnd*s that it returns.

Figure 1 shows these interfaces. Figures 2 and 3 show how these interfaces would overcome the problems described above. Figure 2 is an example of loop that breaks the encapsulation of Person. Figure 3 shows our approach: The access to the properties of Person is moved to methods and the loop is replaced with using the above interfaces. The following sections describe these interfaces in greater detail.

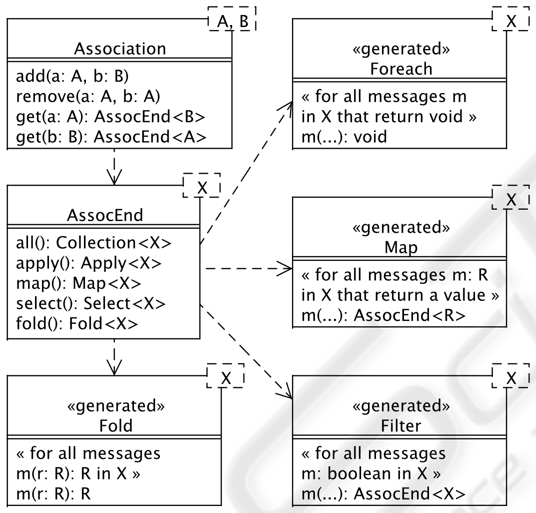

Figure 1: The interfaces for associations.

```
for (Person p: getEmployees()) {
  if (p.geName().getSalary() > limit) {
    p.setSalary(p.getSalary()*0.9);
  }
}
```

Figure 2: An example of the body of the loop that breaks encapsulation.

2.1 The Association Interface

The Association⟨A,B⟩ interface represents the relation itself. It contains the operations for adding and removing pair from the association. The get operations return an AssocEnd. The getter methods of the participating classes, A and B, delegate to the get operations

```
public class Person {
  ...
  public boolean earnsMoreThan(int x) {
    return getSalary() > x;
  }
  public void giveRaise(double d) {
    setSalary(getSalary() * d);
  }
}

getEmployees()
  .filter().earnsMoreThan(limit)
  .foreach().giveRaise(0.9);
```

Figure 3: An encapsulation-preserving approach to associations.

in the association. Figure 4 shows a simple association. Figure. 5 shows the Association interface and its relations with the participating classes.

As mentioned above, the implementation of *Association* requires a decision which concrete *AssocEnd*s to return. This decision can be taken either at design time, by hard-coding it or generating code for it, or at run-time, by using a factory.

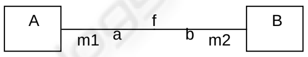

Figure 4: A minimalistic example of an association.

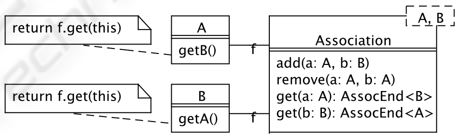

Figure 5: The association interface and participating classes.

2.2 The Association End Interface

The AssocEnd⟨X⟩ interface represent one end of the association. The get operation in Association returns an instance of this interface. This instance represents all the instances of the opposite type that are associated with the given object. The interface has two kinds of operations:

- The operation all returns the associated objects as a platform-specific collection. For example, in our Python implementation from Section 4, it returns a list of all associated objects.
- The operations foreach, map, filter, and fold return an *aggregator*. The aggregator represents operations over the collection of associated instances, without exposing the implementation details. The aggregation interfaces are described below.

As mentioned above, the implementation of *AssocEnd* requires a decision which concrete instances of *foreach*, *map*, *filter* and *fold* to return. These decisions can be taken either at design time, by hard-coding it or generating code for it, or at run-time, by using a factory per aggregator type.

2.3 Aggregation Interfaces

Foreach, *Map*, *Filter*, and *Fold* are *aggregation interfaces*. They are named after higher-order functions from functional programming languages. For example, see the Haskell standard list library[6]. They represent operations over a set of associated instances. The operations in each aggregation interfaces derive from the type of the instances that it wraps. Therefore, these interfaces are *generated* for each type that participates in an association. Figure 6 shows an example of a type and the aggregation interfaces that are derived from it. The details for each interface follow.

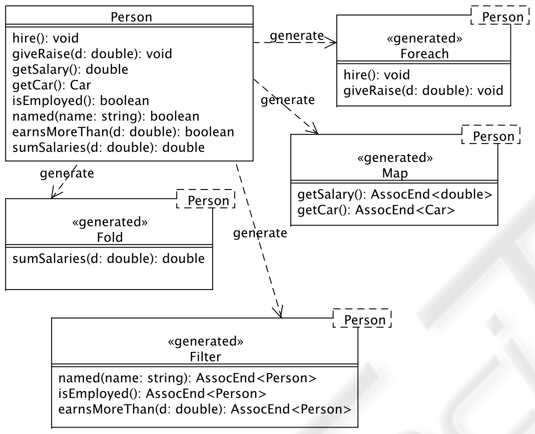
Person
hire(): void
giveRaise(d: double): void
getSalary(): double
getCar(): Car
isEmployed(): boolean
named(name: string): boolean
earnsMoreThan(d: double): boolean
sumSalaries(d: double): double
generate
generate
generate
generate
Person
«generated»
Foreach
hire(): void
giveRaise(d: double): void
Person
«generated»
Map
getSalary(): AssocEnd<double>
getCar(): AssocEnd<Car>
Person
«generated»
Fold
sumSalaries(d: double): double
Person
«generated»
Filter
named(name: string): AssocEnd<Person>
isEmployed(): AssocEnd<Person>
earnsMoreThan(d: double): AssocEnd<Person>

Figure 6: The type Person and generated aggregation interfaces.

2.3.1 Foreach

The *Foreach* aggregator represents sending a message m_i to all the associated instances, where m_i returns no value. The implementation is not required to wait for the return value, so it can be parallel or asynchronous. Figure 7 shows a sequential generated code of a single method in an Foreach aggregator. Applying a message m on the associated instances of a is coded as `a.getB().foreach().m()`.

2.3.2 Map

The *Map* aggregator is similar to *Foreach*, but for messages that has a return value. The map higher-order function is described in “Anatomy of LISP”

[6]www.haskell.org/ghc/docs/latest/html/libraries/haskell98-1.0.1.1/List.html

```
public class ForeachPerson
  implements Foreach<Person>
{
  public void giveRaise(double d) {
    for (Person p: associatedPersons) {
      p.giveRaise(d);
    }
  }
}
```

Figure 7: A sequential implementation of the method giveRaise in Foreach⟨Person⟩.

(Allen, 1978), and was even a part of the APL programming language (Iverson, 1962). It is similar to the OCL collection operation `collect`, see (OMG, 2005) for details. The collection of return values is returned as an aggregator interface itself. Again, the decision which concrete AssocEnd to use can be taken at design-time or run-time. Figure 8 shows a sequential generated code of a single method in a Map aggregator. Since that Map requires the return values, it can run in parallel or send asynchronous messages, but must wait for all the responses to arrive. Getting the mapped values of a message m on the associated instances of a is coded as `a.getB().map().m()`. Aggregated operations can be concatenated. For example, `a.getB().map().m().foreach().f()`.

```
public class MapPerson
  implements Map<Person>
{
  public AssocEnd<Car> getCar() {
    AssocEnd<Car> ret =
      (select a concrete AssocEnd);
    for (Person p: associatedPersons) {
      ret.add(p.getCar());
    }
    return ret;
  }
}
```

Figure 8: A sequential implementation of the method getCar in Map⟨Person⟩.

2.3.3 Filter

The *Filter* aggregator filters associated instances that return `true` in response to a boolean method m_i. It is similar to the OCL collection operation `select`, see (OMG, 2005) for details. Similarly to *Map*, the selected instances are returned as an instance of AssocEnd, whose concrete type is chosen either at design- or run-time. The parametric type of the returned association end is the same as the one of the *Filter* aggregator. Figure 9 shows a sequential generated code of a single method in a *Filter* aggregator. Like *Map*, *Filter* can have a parallel or asynchronous implementation, but it returns only after receiving all the responses. Getting the selected associated instances of a by a boolean message m is coded as `a.getB().filter().m()`.

```
public class FilterPerson
  implements Filter<Person>
{
  public AssocEnd<Person>
  earnsMoreThan(double d)
  {
    AssocEnd<Person> ret =
      (select a concrete AssocEnd);
    for (Person p: associatedPersons) {
      if (p.earnsMoreThan(d) {
        ret.add(p.getCar());
      }
    }
    return ret;
  }
}
```

Figure 9: A sequential implementation of the method earnsMoreThan in Filter⟨Person⟩.

2.3.4 Fold

The *Fold* aggregator performs an operation over the associated instances where each step depends on the result of the previous one.[7] It is similar to the OCL collection operation `iterate`, see (OMG, 2005) for details. The signature of a method that performs such operation is such that it takes as a parameter a value of the same type as it returns: $m_i(r : R) : R$. The folded operation returns the result of the last operation in this chain. The implementation of *Fold* must be sequential. Figure 10 shows an example of a generated code of a single method in a *Fold* aggregator. Getting the final folded result r2 of message m over the associated instances of a with initial value r1 is coded as `R r2 = a.getB().fold().m(r1)`.

```
public class FoldPerson
  implements Fold<Person>
{
  public double sumSalaries(double d) {
    double ret = d;
    for (Person p: associatedPersons) {
      ret = p.sumSalaries(ret);
    }
    return ret;
  }
}
```

Figure 10: A sequential implementation of the method sumSalaries in Fold⟨Person⟩.

[7]The *fold* operation here is from the first elements to the last. The opposite *fold* can be implemented in the same way.

3 IMPLEMENTATION

The Association, AssocEnd, and the aggregator interfaces only specify *what* are the operations over the association, and not *how* to implement them. The concrete implementation may vary in the following aspects:

1. Where to store the relation itself.
2. How to implement the traversal operations of the Foreach and Map aggregators: Sequentially or in parallel.
3. How to implement the selection operations of the Filter aggregator: By traversing the association or using auxiliary data structures.

3.1 Storing the Relation

There are two options for storing the data of the relation:

Internal. The relation is stored in the main memory.

External. The relation is handled by an external component, such as a database or a storage service.

3.1.1 Internal Storage

Storing the relation in main memory, as a part of an OOPL class hierarchy, has been discussed thoroughly in the works mentioned above. Figure 11 shows a design example for implementing an association using classes (in an OOPL). Figure 12 outlines the code for this implementation.

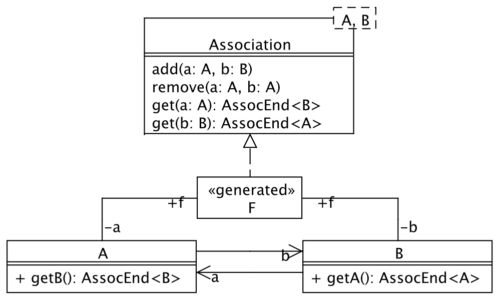

Figure 11: Storing a relation in main memory.

3.1.2 External Storage

The interfaces discussed in Section 2 can act as a uniform façade for persistency components, such as databases or files. As is, the proposed interfaces can not bridge the gap between the object-oriented design and relational databases or sequential files. However, it can provide a common interface for external persistency solutions. For example:

```
public class F
  implements Association<A, B>
{
  public void add(A a, B b) {
    a.b.add(b);
    b.a.add(a);
  }
  public void remove(A a, B b) {
    a.b.remove(b);
    b.a.remove(a);
  }
  public AssociationEnd<B> get(A a) {
    // return an AssociationEnd of a.b
  }
  public AssociationEnd<A> get(B b) {
    // return an AssociationEnd of b.a
  }
}
```

Figure 12: Implementing an association using the fields approach.

- Object-to-Relational Mapping (ORM) systems, e.g. Hibernate[8]
- Object-Oriented Databases (OODB), e.g. DB4O[9]
- Files — See (Blaha and Premerlani, 1997) for Object-to-File serialization techniques

Having a common interface to different solutions allows the developer to switch between solutions seamlessly, thus removing a potential vendor lock. Figure 13 shows an example of hiding the persistency details by using the Association interface: It includes extra code to update a DB4O database.

```
public class F
  implements Association<A, B>
{
  public void add(A a, B b) {
    a.b.add(b);
    b.a.add(a);
  ⇒db.set(a);
  ⇒db.set(b);
  ⇒db.commit();
  }
  public void remove(A a, B b) {
    a.b.remove(b);
    b.a.remove(a);
  ⇒db.set(a);
  ⇒db.set(b);
  ⇒db.commit();
  }
}
```

Figure 13: Implementing an association using DB4O. Marked lines are DB4O-specific.

[8]http://www.hibernate.org

[9]http://www.db4o.com

3.2 Traversal Operations

The iteration over an association does not have to be sequential, as in the examples in Section 2.3. Unlike *Fold* operations, *Foreach*, *Map*, and *Filter* operations do not imply a specific order in which the aggregated message is sent to the associated instances. *Foreach* operations can be sent asynchronously, without waiting for a reply. Figure 14 shows a sequence diagram of performing an *Foreach* operation asynchronously. *Map* and *Filter* operations must wait for returned values, but can still be performed in parallel. Figure 15 shows a sequence diagram of performing a *Map* operation using several threads.

The parallel implementation of traversal operations in similar to the implementation of *parallel iterators*, as described in (Giacaman and Sinnen, 2008a; Giacaman and Sinnen, 2008b). However, parallel iterators and parallel operations differ in several points. First, parallel iterators require changes to the client code. They implements the standard iterator interface, but the programmer still has to add threads to the code. Second, in the cases of *map*, *filter* and *fold*, the programmer has to add a *reduce* method to collect the results. Finally, parallel iterators encourage exposing the structure of the traversed elements, just as sequential ones (see Section 2).

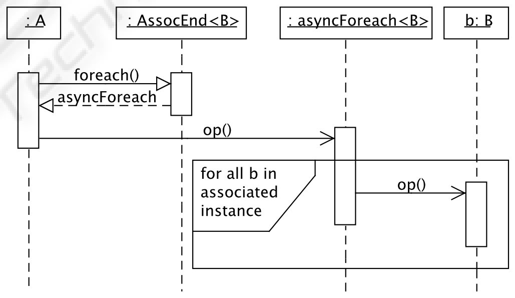

Figure 14: A sequence diagram of an asynchronous *Foreach* operation op(). Asynchronous operations are marked with a non-filled arrow-head.

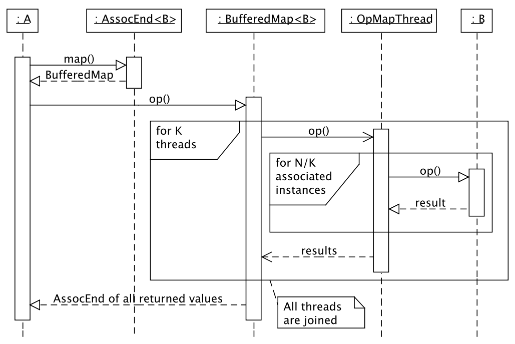

Figure 15: A sequence diagram of a buffered *Map* operation op() using K threads over N instances. Asynchronous operations are marked with a non-filled arrow-head.

3.3 Selecting Instances

A common scenario of traversing over associated instances is finding an instance, or a set of instances, that satisfies a condition — E.g., has a given name. The operations in the *Filter* interface (Section 2.3.3) represent this scenario. Figure 9 shows the most naïve implementation of this scenario.

A common optimization of this scenario is to keep a cache — E.g., a hash-table mapping names to instances with that name. This optimization reduces the run-time in an order of magnitude. Another possible form of cache is to query a database, as discussed in Section 3.1.2.

Implementing this optimization requires adding code to the add and remove operation and change specific Filter operations. Therefore implementing it as a part of the client code is cumbersome and makes the client code dependent of the specific association implementation. A better solution is to generate this code as a part of the implementations of the Association and Filter interfaces. By doing so, it de-couples the client code from the optimization code. Figures 16 and 17 shows the modifications to Association and Filter when caching the property B.key. Figure 18 outlines a similar method that uses DB4O as an external storage and cache.

```
public class F
  implements Association<A, B>
{
  public void add(A a, B b) {
    a.b.add(b);
    b.a.add(a);
    a.cache.put(b.key, b);
  }
  public void remove(A a, B b) {
    a.b.remove(b);
    b.a.remove(a);
    a.cache.remove(b.key, b);
  }
}
```

Figure 16: Caching the property B.key.

```
public class FilterB
  implements Filter<B>
{
  public AssocEnd<B> isKey(Key k) {
    AssocEnd<B> ret =
      (select a concrete AssocEnd);
    ret.add(a.cache.get(key));
    return ret;
  }
}
```

Figure 17: Selecting an instance of B by B.key using the cache.

```
public class FilterB
  implements Filter<B>
{
  public AssocEnd<B> keyed(Key k) {
    AssocEnd<B> ret =
      (select a concrete AssocEnd);
  ⇒Query q = objectContainer.query();
  ⇒q.constrain(Class.B);
  ⇒q.descend("key").constrain(key);
  ⇒ret.add(q.execute());
    return ret;
  }
}
```

Figure 18: Selecting an instance of B by B.key using DB4O. Marked lines are DB4O-specific.

4 EXAMPLE

To demonstrate our scheme and its usage, we provide the Python module `assoc` and code examples. The module includes functions that generates code for associations, using the classes described above.

We decided to use Python in order to avoid the need to generate source code as text. The `assoc` module uses the ability of Python to generate at run-time and return functions and classes as return values. It deals directly with classes, rather then parsers and templates.

4.1 Implementation Details

The `assoc` module works as follows:

1. The class `Assoc_n_n` is constructed with a description of the association and a set of factory functions. The description includes the participating classes and the names of the association ends. The factory functions create the *AssocEnd* objects. Each such factory function takes additional factory functions as arguments. The functions return the aggregators: *Foreach*, *Map*, *Filter* and *Fold*.
2. When created, the association class adds code to the participating classes that implements the association as a pair of list fields. The added code in each class includes:
   a. A wrapper to the constructor that initialize the relation data and the *AssocEnd* object.
   b. A getter for the association end.
3. The association object uses the factory functions to return association ends and aggregators.
4. The *AssocEnd* and aggregator objects are initialized with a reference to a list of associated instances.

5. Each *AssocEnd* object has `onAdd` and `onRemove` methods, to handle special cases, such as caching (see Section 3.3).

The module supports the following aggregators:

- Serial traversal over associated instances.
- Parallel traversal with *K* threads over associated instances.
- Maintaining a cache based on a given key function and filtering instances using this cache.

4.2 Usage Example

Figure 19 shows two Python classes that represents workers in a department. We would like to implement an association between the department and its employees, so that each department will hold a set of its workers. Figure 20 shows the code for creating the association object with default factory function. The command `Emps.add(dept, emp)` will associate the given department and employee.

```
class Dept(object): pass

class Employee(object):
  def __init__(self, id, name):
    self.id = id
    self.name = name
  def getId(self):
    return self.id
  def hasId(self, id):
    return self.id == id
  def getName(self):
    return self.name
```

Figure 19: Associated classes in Python.

```
Emps = Assoc_n_n(
 cls1=Dept, cls2=Employee,
 name1='workers', name2='worksIn',
 factory1=ListAssocEndFactory(Employee),
 factory2=ListAssocEndFactory(Dept))
```

Figure 20: An association object in Python.

When the association object is created, it add the participating classes getter methods for the association ends. In this case, it add the method `getWorkers()` to the class `Dept`, and `getWorksIn()` to `Employee`. The *AssocEnd* objects provide the aggregator operations. For example, the function in Figure 21 returns a worker in a department by id.

The association in Figure 19 uses the default implementations. Therefore, the aggregator operations are performed by iterating over the list of associated objects sequentially. Finding an employee with a given id by scanning the entire list can slow down the application if a department has many employees.

```
def deptWorkerById(dept, id):
  return dept.getWorkers()
             .filter().hasId(id)[0]
```

Figure 21: Getting a worker in a department by id.

It is possible to solve this problem by replacing the default factory function for filter operations. The association shown in Figure 22 uses a *Filter* operation with caching. As expected, our measurements shows that the function from Figure 21 runs faster by an order of magnitude with these settings. Note that no change in this function or in the above classes is required to achieve this boost in performance. Similarly, Figure 23 shows an association that uses multiple threads for some aggregator operations. These settings can speed up batch I/O operations (for example, each employee object writes something to a file), also without changing any client code.

```
Emps = Assoc_n_n(
 cls1=Dept, cls2=Employee,
 name1='workers', name2='worksIn',
 factory1=ListAssocEndFactory(Employee,
  filterImpl=makeListQualFilter(
   Employee.getId, Employee.hasId)),
 factory2=ListAssocEndFactory(Dept))
```

Figure 22: An association with caching for employee ids.

```
Emps = Assoc_n_n(
 cls1=Dept, cls2=Employee,
 name1='workers', name2='worksIn',
 factory1=ListAssocEndFactory(Employee,
  foreachImpl=makeListSpawnForeach(5),
  mapImpl=makeListSpawnMap(5)),
 factory2=ListAssocEndFactory(Dept))
```

Figure 23: An association with 5 threads for *Foreach* and *Map* operations.

This module with its test code is available from http://www.cs.bgu.ac.il/∼gwiener/software/associations.

5 CONCLUSIONS

In this work we showed a novel approach for generating code for associations. Rather than focusing on a single, sequential, in-memory implementation, we presented a flexible and open approach. Our approach is based on a set of interfaces that provide operations over the entire relation. These operations include managing the relation through the *Association* interface, and traversing it using *Foreach*, *Map*, *Fil-*

ter, and *Fold*. Traversals, excluding *Fold*, can be implemented by a multi-threaded code. The *Association* and *Filter* interfaces encapsulate optional optimizations, such as caching or using a database. By providing this encapsulation, the client code is independent of the implementation details of the association.

5.1 Future Work

Our current work covers only the basic functionality of associations — I.e., to operate on the objects that are associated with a given instance. It does not cover the following advanced topics: 1) Association classes 2) Properties of associations, such as set and ordered 3) Relations between associations, such as sub-set and override. We plan to include these topics in the following iterations of our work. Specifically, we intend to support relations between associations in our scheme, so that adding or removing these relations will not affect the client code.

ACKNOWLEDGEMENTS

This work has been prepared with the kind support of the Frankel Center for Computer Science at the Ben-Gurion University.

REFERENCES

Akehurst, D., Howells, G., and McDonald-Maier, K. (2007). Implementing associations: UML 2.0 to Java 5. *Software and Systems Modeling*, 6(1):3–35.

Allen, J. (1978). *Anatomy of LISP*. McGraw-Hill Book Company.

Blaha, M. and Premerlani, W. (1997). *Object-oriented modeling and design for database applications*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

Bloch, J. (1995). The Java Tutorials: Collections. http://java.sun.com/docs/books/tutorial/collections.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gessenharter, D. (2008). Mapping the UML2 semantics of associations to a Java code generation model. In Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., and Völter, M., editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 813–827. Springer.

Giacaman, N. and Sinnen, O. (2008a). Parallel iterator for parallelising object-oriented applications. Technical Report 669, University of Auckland.

Giacaman, N. and Sinnen, O. (2008b). Parallel iterator for parallelising object oriented applications. In *Proceedings of the 7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, pages 44–49. World Scientific and Engineering Academy and Society (WSEAS).

Harrison, W., Barton, C., and Raghavachari, M. (2000). Mapping UML designs to Java. *ACM SIGPLAN Notices*, 35(10):178–187.

Iverson, K. E. (1962). *A Programming Language*. John Wiley & Sons, Inc.

Larman, C. (2004). *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Prentice Hall PTR Upper Saddle River, NJ, USA.

Martin, J. and McClure, C. (1983). *Software Maintenance: The Problems and Its Solutions*. Prentice Hall Professional Technical Reference.

Martin, J. and Odell, J. J. (1992). *Object-oriented analysis and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

OMG (2005). OCL 2.0 Specification. *Formal specification, Object Management Group*.

Pigoski, T. M. (1996). *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., New York, NY, USA.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.