Migration from Manual Maintenance to Automatic Generation of an Information Model

Konstantin Läufer

Department of Computer Science, Loyola University Chicago,
6525 N. Sheridan Road, Chicago, IL 60626, USA
laufer@cs.luc.edu
http://www.cs.luc.edu/~laufer/

Abstract. In the context of developing the next major release of a commercial infrastructure product for application integration, we describe our vision for and experience with the migration from a conventional, manually maintained, language-specific information model implementation toward a model-driven, language-independent solution that allows the implementation and other artifacts to be generated automatically. Additional requirements include backward compatibility and better support for the existing development process, as well as new features such as the ability to express constraints in the model. We rely on reverse engineering to propagate existing domain knowledge from the old implementation to the new model. We have implemented a prototype including both reverse engineering and automatic generation as extension modules of a commercial integrated development platform. The key turned out to be the incremental co-development of the reverse-engineering tool and the generator. Our experience has been positive, and we argue that it can be generalized to similar information model migration scenarios.

1 Introduction

In the enterprise software industry, most systems rely on some sort of information model. Evolving system requirements often entail changes to the information model, regardless of the specific application domain or system architecture. Therefore, the ability to change the information model rapidly and reliably is crucial from a business perspective.

In this paper, our starting point is a conventional, language-specific, object-oriented information model whose implementation is maintained manually. We describe our experience migrating to our envisioned end point, a model-driven, language-independent solution that allows the implementation and other artifacts to be generated automatically [7]. As a result, we expect the turnaround time and reliability of information model maintenance to improve significantly. We argue that our experience can be generalized to other systems based on object-oriented information models, especially when implementation decisions and other knowledge embedded in legacy components must be preserved during the migration to a model-driven solution.

The context for our work is the development of the next major release of a commercial infrastructure product for application integration. The high-level architecture of the product is quite typical in the area of enterprise application integration [3]. The main purpose of the product is to provide connectivity between applications (web, wireless, or voice-based) and data sources, as well as some services such as authentication. Conceptually, the infrastructure is parameterized by and largely oblivious to a domain-specific information model. The infrastructure allows the exchange of information either point-to-point or in a routed fashion; the data itself is serialized to a proprietary XML representation or some other suitable form. An overview of this architecture can be found in Fig. 1.

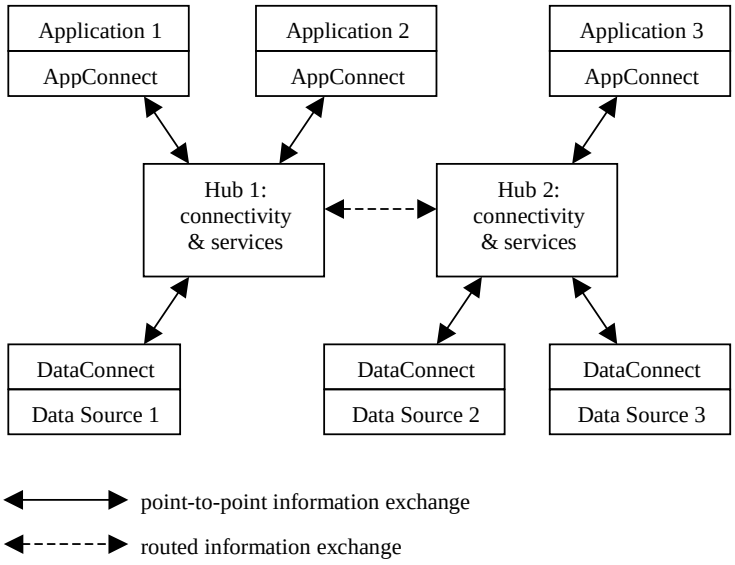

Fig. 1. High-level architecture of connectivity solution

Within the company, the impending development of a new system release was seen as an opportunity to improve the information model architecture and maintenance process. The specific objectives of the envisioned changes are as follows:

- Automate the implementation process.
- Automate the generation of other derived artifacts.
- Achieve language independence to support multiple target platforms, for example, J2EE [11] and .NET [5].
- Eliminate the current gap between the analysis and design models in an effort to enable business analysts to conduct information modeling.

Several challenges obstruct the path toward this vision, including knowledge represented outside the API, inconsistencies in the API, and insufficient UML expertise of

business analysts. In this paper, we describe our experience in coming up with a practical solution to these challenges.

In the remainder of the paper, we first discuss the existing approach and its shortcomings, and we outline our vision of a solution. Then, we elaborate on the practical challenges that we face on our way to the envisioned solution. Finally, we develop a solution that addresses these challenges.

2 The Existing Approach and its Shortcomings

In this section, we describe the existing information model architecture and the process used for changes in the information model. We then discuss the various problems with this approach.

The Information Model Architecture

The information model API in our system is a collection of interfaces related by the Composite pattern [3]. In our case, the information model is essentially a passive data structure, that is, it has only attributes corresponding to back-end data fields or subobjects; other than the methods to access and modify these attributes, the information model provides no true operations. An abstract factory is used to create instances of implementation classes of these interfaces. The application programmer then programs against this API. The API is currently available for the Java platform.

Concretely, the information model implementation classes have a common superclass called `Base` that defines an associative array in which all attributes are stored; the vast majority of the accessor and mutator methods simply hide the interaction with the associative array. Other kinds of methods, such as accessors with additional logic, are rare.

There are three kinds of attributes in the data model. Each abstract attribute `attr` of type `Type` is represented by one or more corresponding methods, depending on its kind:

- Read-only single-valued attribute: `Type getAttr()`.
- Read-write single-valued attribute: `Type getAttr()` and `void setAttr(Type newValue)`.
- Multi-valued attribute: `Type[] getAttr()`, void `setAttr(Type[] newValues)`, `void addAttr(Type newValue)`, and `void removeAttr(Type value)`.

Accessor methods for Boolean attributes are prefixed with "`is`" instead of "`get`". Although these attributes are equivalent to properties in the Java Beans framework [12], the Java Beans convention was not followed for historical reasons. By convention, all interfaces in our API start with upper-case "`I`". The API also contains some concrete classes to represent enumerated types.

The following example shows an interface with a single-valued attribute `address` and a multi-valued attribute `statement`, followed by a suitable implementation

class. The string constants in the implementation class serve as keys into the associative array and element tags for externalization as XML.

```
interface IAccount extends Base {
  IAddress getAddress();
  void setAddress(IAddress address);
  IStatement[] getStatement();
  void setStatement(IStatement[] statement);
  void addStatement(IStatement statement);
  void removeStatement();
  // …
}

class Account extends Base implements IAccount {
  private static final String ACCT_ADDR = "ACCT_ADDR";
  private static final String ACCT_STMT = "ACCT_STMT";
  public IAddress getAddress() {
    return (IAddress)getItem(ACCT_ADDR);
  }
  public void setAddress(IAddress address) {
    setItem(ACCT_ADDR, address);
  }
  public IStatement[] getStatement() {
    return(IStatement[])getList(IStatement.class, ACCT_STMT);
  }
  public void setStatement(IStatement[] statement) {
    setList(ACCT_STMT, statement);
  }
  public void addStatement(IStatement statement) {
    addToList(ACCT_STMT, statement);
  }
  public void removeStatement() {
    setList(ACCT_STMT, null);
  }
  // …
}
```

The Existing Information Model Maintenance Process

Let us now take a look at the existing information model maintenance process in terms of its roles and artifacts. This process is a subprocess of an enterprise-wide adaptation of the Rational Unified Process (RUP) framework [9]; accordingly, most artifacts in this process are represented in the Unified Modeling Language (UML). The following steps outline the process and, implicitly, describe the roles involved and the artifacts they own. Fig. 3 illustrates the process visually.

1. The *business analyst* produces *use cases* for new or updated application or infrastructure requirements to be supported. Information model changes are usually driven by new or changed requirements at the application or infrastructure level. This step is repeated until the use cases pass a joint review by the business analyst and the *application developer*.
2. The *information modeler* produces an *analysis model* based on each use case. This step is repeated until the analysis model passes a joint review by the business ana-

lyst and the information modeler. The analysis model is comprised of a high-level class diagram to show the attributes and links of the classes for the use cases and, if needed, sequence diagrams to indicate how to use instances of the classes according to the steps of the use case. An example of such a class diagram can be found in Fig. 2.

3. The information modeler produces a *design model* based on the analysis model. This step is repeated until the design model passes a joint review by the information modeler and the application developer. The design model is represented as a collection of Java interfaces or its equivalent UML class diagram. The `IAccount` interface shown above is part of the design model.
4. The *infrastructure developer* produces an *implementation* of the design model along with a standard unit test for each resulting class. This step is repeated until the implementation passes the associated unit tests. The implementation is a collection of Java classes. For example, the `Account` class shown above is part of the implementation.
5. The business analyst manually creates or updates the *data dictionary* to reflect any changes in the design model. The data dictionary serves as documentation of the information model suitable for non-programmers.

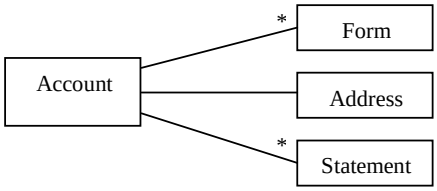

Fig. 2. Sample portion of the analysis model

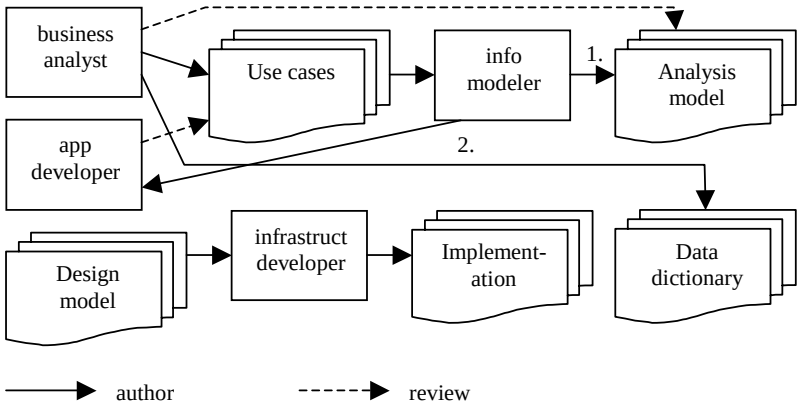

Fig. 3. Existing information modeling process

Shortcomings of the Existing Process

The existing process has several substantial shortcomings that limit the productivity of the organization and the quality of the product.

- The manual implementation process is tedious and repetitive, making it error-prone and wasteful of valuable infrastructure developer time. This part of the process has a very high potential for automation.
- The design model is too low-level. It consists of Java interfaces with explicit methods instead of attributes and associations. Consequently, there is a larger gap than necessary between the analysis model and the design model, which makes the information modeler's job harder than necessary and prone to errors.
- The design model is language-specific. This stands in the way of supporting multiple enterprise platforms, such as J2EE and .NET.
- Other artifacts such as data dictionaries, XML schemas, and documentation other than Javadoc are created and maintained by hand. This is also tedious and wastes valuable business analyst and technical writer resources.
- The present API is limited by the expressiveness of Java interfaces. This means that behavioral information about methods is limited to method signatures. Additional information such as pre- and postconditions cannot be expressed.

3 Envisioned Solution

In this section, we describe our envisioned solution in terms of its process and supporting architecture. We envision a model-driven approach, using a sufficiently abstract model as the single source artifact. All other artifacts, in particular the information model implementation, will be generated from the abstract model for multiple target languages or platforms. Our solution is driven by the objectives stated in section 1. The envisioned high-level information modeling process is as follows. The process is also illustrated in Fig. 5.

1. With the help of the information modeler, the business analyst creates an abstract information model based on the requirements. The abstract model consists of classes with attributes and associations, but no accessors or mutators; each attribute or association represents a concrete attribute or subobject. This kind of abstract model is platform- and language-independent. An example of an abstract model is shown in Fig. 4.
2. From this abstract model, a concrete implementation model is created automatically. The implementation model consists of a concrete API and its implementation classes. Each attribute in the abstract model is expanded to a suitable collection of accessor and mutator methods, whose exact nature depends on the specifics of the attribute. The concrete physical layer is platform- and language-specific.
3. Other artifacts will be generated from the abstract model as well, including the following:
   - Unit tests

- Data dictionary
- XML schemas
- List of changes from previous version for release notes

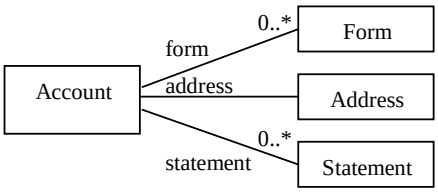

Fig. 4. Sample abstract model. The association names are the attribute names, which later become part of the generated method names

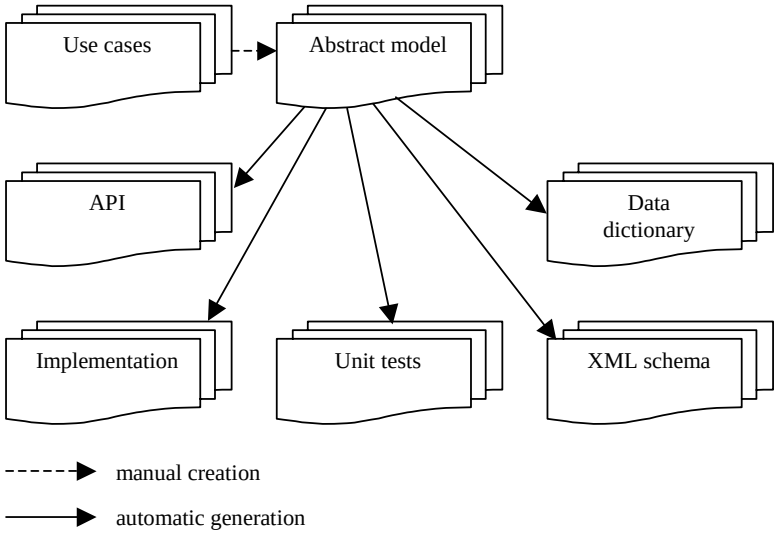

Fig. 5. Envisioned information modeling process

4 Practical Challenges

In this section, we assess the practicality of the vision outlined in the previous section. Several challenges stand in the way of an easy migration from the current to the envisioned information modeling process:

– Substantial knowledge related to the information model is represented outside the API, embedded in artifacts that we want to be generated, most notably the information model implementation. As part of the migration, all such knowledge has to be captured and propagated into the new abstract model.
– There are numerous inconsistencies in the API, which cannot simply be fixed because of backward compatibility required to support the installed client base.
– The business analysts do not typically have sufficient command of UML to produce an abstract information model such as the one in Fig. 4.

We now address each of these challenges.

Knowledge Represented Outside the API

The most serious challenge in implementing the envisioned process is the significant amount of knowledge represented outside the information model API, above all, implementation decisions currently embedded in the information model implementation. All such knowledge must be identified and incorporated in the abstract model.

Specific knowledge embedded in the information model implementation includes the following:

– Most methods are implemented in standard ways using the methods in the common `Base` class, as illustrated in the example of the `Account` class in section 2. However, some method implementations do not conform to this standard because they contain additional logic. All nonstandard method implementations must be identified and propagated to the abstract model in some form. There are currently about 30 methods in this category.
– The tag names for the externalized XML representation of an object are defined as String constants in the corresponding implementation class.

The following class illustrates such embedded knowledge:

```
public class Person extends Base implements IPerson {
  private static final String PERSON_ACCT = "PERSON_ACCT";
  // …
  public void addAccount(IAccount newValue) {
    // Make the account's key the same as the person's.
    newValue.setKey(getKey());
    addToList(PERSON_ACCT, newValue);
  }
}
```

Given the size of the information model API–about 250 interfaces and about 100 classes representing enumerated types–it is not practical to do migrate this knowledge manually.

Inconsistencies in the API

There are numerous inconsistencies in the API that have to be considered in the migration process. Furthermore, the backward compatibility requirement gets in the way of simply fixing the inconsistencies.

The most common kind of inconsistency is incorrectly named methods. Most attributes in our information model correspond to collections of methods in systematic ways (see also section 2), similar to the Java Beans naming convention for properties. However, there are numerous inconsistencies with this convention, especially in the case of multi-valued data points. For example, instead of methods `getAttr` and `setAttr` (singular), there might be methods `getAttrs` and `setAttrs` (plural) along with `addAttr` and `removeAttr`. There are about 25 multi-valued attributes in the API with inconsistent method names. The attribute `form` in the `IAccount` interface illustrates this problem:

```
interface IAccount extends Base {
  // …
  void addForm(IForm form);
  void setForms(IForm[] forms);
  IForm[] getForms();
  void removeForm(IForm form);
}
```

Furthermore, there are many Boolean methods that do not follow the convention of the prefix "`is`". All methods that are associated with an attribute must be recognized and grouped together, and their inconsistent naming must be represented in the abstract model for later re-generation of the original method names. Eventually, inconsistent methods should be deprecated and replaced by consistent ones. There are currently about 50 Boolean methods in this category.

Other inconsistencies include interfaces, such as `I1040Form`, whose translated implementation class names (by dropping the initial "`I`") would not be valid identifiers and, therefore, have to be treated as exceptions. There are about 10 misnamed interfaces.

Some methods have nonstandard signatures as a result of the wrong level of abstraction. For example, the method `addPhone` in the following class should have had a single argument of type `IPhoneEntry`. There are about 35 such cases.

```
public class Person extends Base implements IPerson {
  private static final String PERSON_PHONE = "PERSON_PHONE";
  // …
  public void addPhone(Phone type, String newValue) {
    if(type != null && newValue != null) {
      IPhoneEntry number = …;
      addToList(PERSON_PHONE, number);
    }
  }
}
```

Finally, some methods are auxiliary methods to support nonstandard method implementations; such methods are part of the implementation but not the API. There are about 40 such cases.

Insufficient UML Expertise of Business Analysts

The typical business analyst writes use cases in the form of text documents. He or she does not have sufficient command of UML to produce an abstract information model

such as the one in Fig. 4. However, we expect business analysts to participate in the new process under the guidance of the information modelers and with appropriate tool support. In the longer term, it is also possible to train business analysts in UML analysis modeling.

5 Toward a Practical Solution

In this section, we address the various challenges described in the previous section in the form of a practical solution. The key insight from the previous section is recognizing that, given the size of the API, it is impractical to capture any anomalies manually, including knowledge outside the abstract model and inconsistencies in the existing interface. Therefore, we turn to reverse engineering and automatic testing to facilitate this part of the migration process.

We first describe an augmented abstract model capable of representing those anomalies. We then discuss tools to reverse-engineer an abstract model from an existing API and implementation, and to generate a new API and implementation from an abstract model. We also discuss how other artifacts can be generated and how language independence could be achieved.

Augmented Abstract Model

We first need to come up with an augmented abstract model that is capable of representing the knowledge and anomalies discovered in section 4:

- Read-only versus read-write attributes
- Nonstandard target class names
- XML tag names
- Incorrect method names
- Nonstandard method implementations
- Methods with nonstandard signatures
- Auxiliary methods

Integrated UML modeling and development platforms typically support user-defined custom properties at different levels of the model, which are a suitable way to represent such information. Custom properties can be single-valued or multi-valued, string or Boolean properties. We represent each piece of additional information as a property at the appropriate level. Specifically, nonstandard target class names become a class-level property, while XML tags, read-only/read-write, and incorrect method names become attribute-level properties.

Representation as string or Boolean properties is appropriate for most of the information we intend to capture. However, some of the information actually includes method bodies. By representing source code as strings, we would give up the support of the development environment for future modification of that code.

Instead, we represent implementations of nonstandard and auxiliary methods by placing them in companion classes to the abstract model classes. The implementation

generator can then generate the standard methods from the interface and copy the nonstandard ones from the companion class. This allows the nonstandard method code to be developed with the full support of the development environment. The companion class would be a stripped-down version of the original implementation class, extending the same superclass and implementing the same interfaces, but containing only the nonstandard methods. In the case of mutually dependent interfaces, the companion classes refer to each other instead of the original implementation classes.

For example, the companion class to the `IPerson` class, whose implementation has a nonstandard method, looks as follows:

```
public abstract class PersonCompanion
    extends BaseCompanion implements IPerson {
  public void addAccount(IAccount newValue) {
    newValue.setKey(getKey());
    addToList(PERSON_ACCT, newValue);
  }
  // …
}
```

Development Platform Support

The Together ControlCenter integrated development platform [13], which is used in this project, exposes an abstract syntax tree representation of all interfaces and implementation classes in the form of two open APIs: an abstract, UML-level API and a concrete, source-level API. In both APIs, abstract syntax trees are represented using the Composite and Interpreter patterns with suitable Visitor support for traversal [3]. All parsing and some code generation are taken care of by the development platform. Most of the work done by the reverse engineering tool and the code generator can be done at the level of the abstract API. Other development platforms have similar APIs.

Reverse Engineering Tool

For each interface in the existing information model API, the reverse engineering tool analyzes the interface and its implementation class together, using simple heuristics to identify methods that belong to the same attribute, to determine the properties of the attribute, and to detect implementation anomalies. The reverse engineering tool creates a class in the new abstract model for this interface and, if necessary, a companion class to capture implementation anomalies.

As an example without implementation anomalies, the reverse engineering tool would transform the `IAccount` interface and the `Account` implementation class shown in section 2 to the diagram in Fig. 2, setting suitable properties of the associations. As an example with implementation anomalies, the tool would transform `IPerson` and `Person` to an abstract `Person` class and the `PersonCompanion` class shown above.

Implementation Generator

For each class in the abstract information model, the implementation generator generates an interface in the API and an implementation class. The generator expands each attribute to a collection of methods, guided by the custom properties of the attributes. If a companion class is present, then the generator uses the implementation information from the companion class to re-generate any nonstandard method implementations.

Incremental Co-development of Reverse Engineering and Code Generation Tools

We intend to minimize any need for manual work. Specifically, we want to avoid manually reviewing the source code of the information model implementation in search for anomalies. Instead, we take the following incremental approach.

- We build an initial version of the reverse engineering tool, which understands only the normal cases of methods corresponding to attributes. While this version does not have the ability to process any anomalies, it is able to flag any anomalies it finds. We can then inspect the flagged anomalies and augment the next version of reverse engineering tool with the capability to handle them. We proceed incrementally, analyzing one package at a time. Eventually, the reverse engineering tool will have an understanding of all anomalies in the entire information model implementation.
- We then build an initial version of the implementation generation tool. Since we already have a list of all anomalies and a way to represent them in the abstract information model generated by the reverse engineering tool, we can provide the generator with the ability to recreate the anomalies from the abstract model.
- We now verify the correctness of the tools and the reverse-engineered abstract model by comparing the original implementation with the generated one by applying the original unit tests to both.

We repeat this process until the system reaches a stable state, in which the generated API and implementation behaves in the same way as the original one [10]. At that point, we can dispose of the original API and implementation and use the new abstract model as our single source for generating the API, implementation, and other artifacts. The process is illustrated in Fig. 6.

Generating Other Artifacts

As envisioned in section 3, it is beneficial to generate other artifacts related to the information model instead of maintaining them manually. In some cases, it is necessary to review whether any information in the artifact must be captured and propagated into the abstract information model.

- XML schemas can be generated by exporting the UML model as XMI [6] and running it through an XML schema generation tool [1]. XML schemas are useful for publishing portions of the information model for communication between web

services (the internal XML tags for the information model classes are proprietary). Other transformation tools can be applied at the XMI level as well [2].

- Several types of documentation can be generated for different target audiences, including application developers, consultants, marketing people, etc. Data dictionaries and lists of changes to the information model are examples in this category.
- Unit test drivers can be generated by analyzing the existing, manually created unit test drivers and augmenting the code generator with the capability to re-create the drivers systematically.

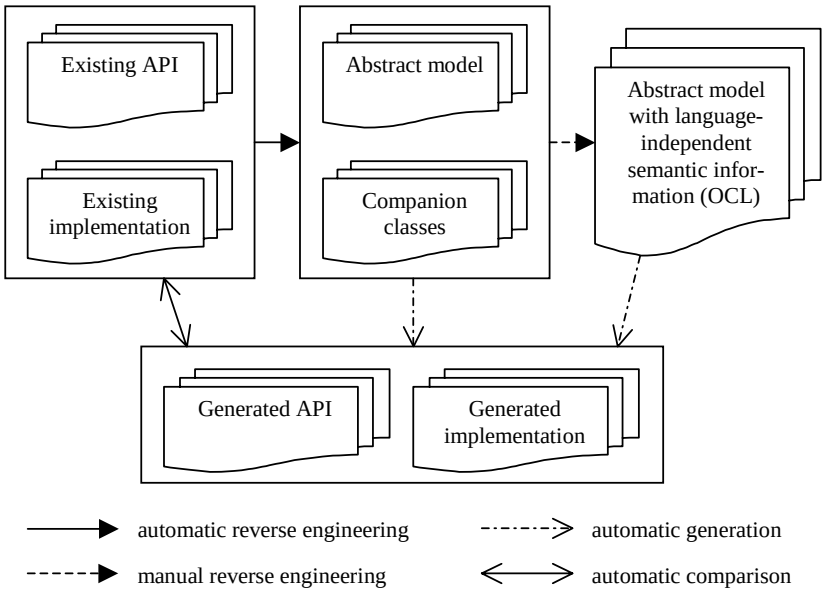

Fig. 6. Migration process: we incrementally co-develop the reverse engineering and implementation generation tools until the generated implementation behaves as the original

Toward Language Independence

Up to this point, language independence has not been achieved. Companion classes corresponding to classes in the abstract model contain code in specific languages. To bridge the remaining gap, any program logic would have to be moved from the companion classes into the abstract model itself and expressed in a language-independent fashion.

The Object Constraint Language (OCL), a part of UML, provides a language-independent mechanism for expressing program logic in functional style within the model [8]. Fig. 7 shows an example of an abstract model in which an implementation anomaly is expressed as an OCL constraint on an association.

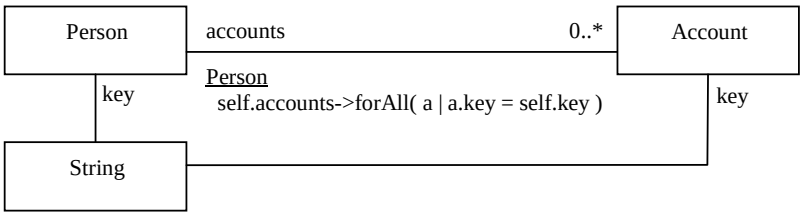

Fig. 7. An abstract model with an implementation anomaly expressed as an OCL constraint

6 Conclusion

In this section, we report on the status of our work and summarize our findings.

At present, we have implemented a prototype of the language-specific portion of the solution described in section 5. The prototype is capable of reverse-engineering the existing information model implementation into an abstract information model, and of re-generating a new implementation from the abstract model. The generation of additional artifacts or unit tests is not supported yet. The prototype consists of about 2000 lines of Java code.

Since the company no longer has the resources to continue product development, it has recently shifted its focus exclusively toward consulting. Consequently, the future of the entire project is uncertain at this point. Based on historical project data for information model maintenance, we would have expected the proposed approach to require about one third less effort than the existing process; the up-front development of the reverse-engineering and code-generation tools would have paid off within about half a year.

Our key experiences can be summarized as follows:

- The automatic generation of an implementation from an abstract model is feasible and promising.
- The incremental co-development of the reverse engineering tool and implementation generator makes it possible to minimize any manual analysis of source code or model.

Finally, we argue that our experience can be generalized to similar scenarios where legacy components are replaced by components that are automatically generated from a model, in particular, when implementation decisions and other knowledge are embedded in the legacy components and must be preserved.

References

1. Carlson, D.: Modeling XML Applications with UML. Practical e-Business Applications. Addison-Wesley (2001).

2. Demuth, B., Hussmann, H., Obermaier, S.: Experiments With XMI Based Transformations of Software Models. http://citeseer.nj.nec.com/465390.html.
3. Fowler, M.: Enterprise Application Architecture. http://www.martinfowler.com/isa/.
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995).
5. Microsoft: The Microsoft .NET Platform. http://www.microsoft.com/net/.
6. Object Management Group: XML Metadata Interchange (XMI), Version 1.2. http://www.omg.org/technology/documents/formal/xmi.htm.
7. Object Management Group: Model Driven Architecture. http://www.omg.org/mda/.
8. Object Management Group: Object Constraint Language Specification, Version 1.1 (1997). http://www-4.ibm.com/software/ad/library/standards/ocl.html.
9. Rational Software: The Rational Unified Process for Systems Engineering (2001). http://www.rational.com/media/whitepapers/TP165.pdf.
10. Rugaber, S., Shikano, T., Stirewalt, K.: Adequate Reverse-Engineering. Proc. Automated Software Engineering Conf. (2001).
11. Sun Microsystems: The Java 2 Platform, Enterprise Edition. http://java.sun.com/j2ee/.
12. Sun Microsystems: The JavaBeans Component Architecture. http://java.sun.com/products/javabeans/.
13. TogetherSoft: Together ControlCenter 5.5. http://www.togethersoft.com/products/controlcenter/.