Z
The SPOOL Design Repository: Architecture, Schema, and Mechanisms

Reinhard Schauer
Rudolf K. Keller
Bruno Laguë
Gregory Knapen
Sébastien Robitaille
Guy Saint-Denis

Z.1 Introduction

The landscape of reverse engineering is rich in tools for the recovery, quantification, and analysis of source code. Most of these tools, however, cover only a small slice of what the notion of *reverse engineering* promises: "a process of analyzing a subject system to (a) identify the system's components and their interrelationships and (b) create representations of a system in another form at a higher level of abstraction" (Chikofsky and Cross, 1990). These tools take hardly into account that reverse engineering is a process of collaborating activities, rather than a focused task of investigating some specific software property. To be effective, the process of reverse engineering demands that tools communicate and that infrastructure support be provided for their coordination. In the SPOOL project, we have developed and integrated a suite of tools in which each tool addresses a different task of reverse engineering yet allows for easy transfer of the gathered information to other tools for further processing (cf. Chapter Y). At the core of such tool collaboration lies the SPOOL repository.

The SPOOL reverse engineering environment (Figure 1) uses a three-tier architecture (Tsichritzis and Klug, 1978; Fowler, 1997) to achieve a clear separation of concerns between the end user tools, the schema and the objects of the reverse engineered models, and the persistent datastore. The lowest tier consists of an *object-oriented database management system* which provides the physical, persistent datastore for the reverse engineered source code models and the design information. The middle tier consists of the *repository schema*, which is an

object-oriented schema of the reverse engineered models, comprising structure (classes, attributes, and relationships), behavior (access and manipulation functionality of objects), and mechanisms (higher-level functionality, such as complex object traversal, change notification, and dependency accumulation). We call these two lower tiers the SPOOL *design repository*. The upper tier consists of end user tools implementing domain-specific functionality based on the repository schema. This includes tools for the parsing of the source code and its transformation into the repository schema (*source code capturing*) as well as tools for the *visualization and analysis* of the source code models. Refer to Chapter Y for details about the upper tier and for information on the research that we conducted based on the SPOOL reverse engineering environment.

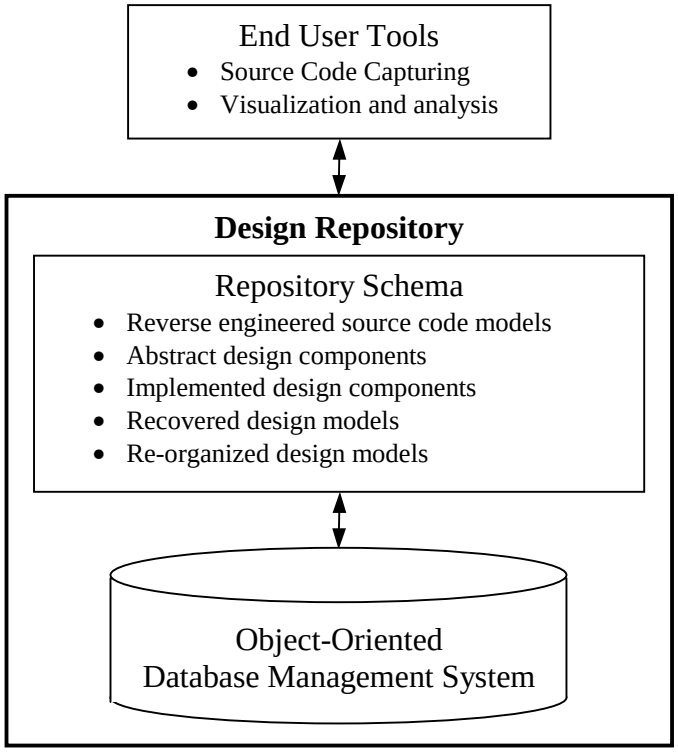

Figure 1. Overview of SPOOL reverse engineering environment.

At the core of the SPOOL reverse engineering environment is the SPOOL *design repository*, which consists of the repository schema and the physical data store. The repository schema is an object-oriented class hierarchy that defines

the structure and the behavior of the objects that are part of the reverse engineered *source code models*, the *abstract design components* that are to be identified from the source code, *the implemented design components*, and the *recovered and re-organized design models*. Moreover, the schema provides for more complex behavioral mechanisms that are applied throughout the schema classes, which includes uniform traversal of complex objects to retrieve contained objects, notification to the views on changes in the repository, and dependency accumulation to improve access performance to aggregated information. The schema of the design repository is based on an extended version of the *UML metamodel 1.1* (UML, 1997). We adopted the UML metamodel to the end of reverse engineering as it captures most of the schema requirements of the research activities of SPOOL. This extended UML metamodel (or SPOOL repository schema) is represented as a *Java 1.1* class hierarchy, in which the classes constitute the data of the *MVC-based* (Buschmann et al., 1996) SPOOL reverse engineering environment.

The object-oriented database of the SPOOL repository is implemented using *POET 6.0* (Poet, 2000). It provides for data persistence, retrieval, consistency, and recovery. Using the precompiler of POET 6.0's *Java Tight Binding*, an object-oriented database representing the SPOOL repository is generated from the SPOOL schema. As POET 6.0 is ODMG 3.0 (ODMG, 2000) compliant, its substitution of POET 6.0 for another ODMG 3.0 compliant database management system would be accomplishable without major impact on the schema and the end user tools.

In the remainder of this chapter, we first detail the architecture of the SPOOL repository. Next, we explain the SPOOL repository schema and its relation to the UML metamodel, discussing the schema's top-level, core, relationship, behavior, and extension classes. Furthermore, we describe three of the key mechanisms of the repository, that is, the traversal of complex objects, model/view change notification, and dependency management. Finally, as a conclusion, we reflect on the use of the UML metamodel as the basis of the SPOOL repository schema, and provide an outlook into future work.

Z.2 Repository Architecture

The major architectural design goal for the SPOOL repository was to make the schema resilient to change, adaptation, and extension, in order to address and accommodate easily new research projects. To achieve a high degree of flexibility, we decided to shield the implementation of the design repository completely from the client code that implements the tools for analysis and visualization. The retrieval and manipulation of objects in the design repository is accomplished via a hierarchy of public Java interfaces, and instantiations and initializations are implemented via *an Abstract Factory* (Gamma et al., 1995). Figure 2 illustrates the architecture of the SPOOL design repository.

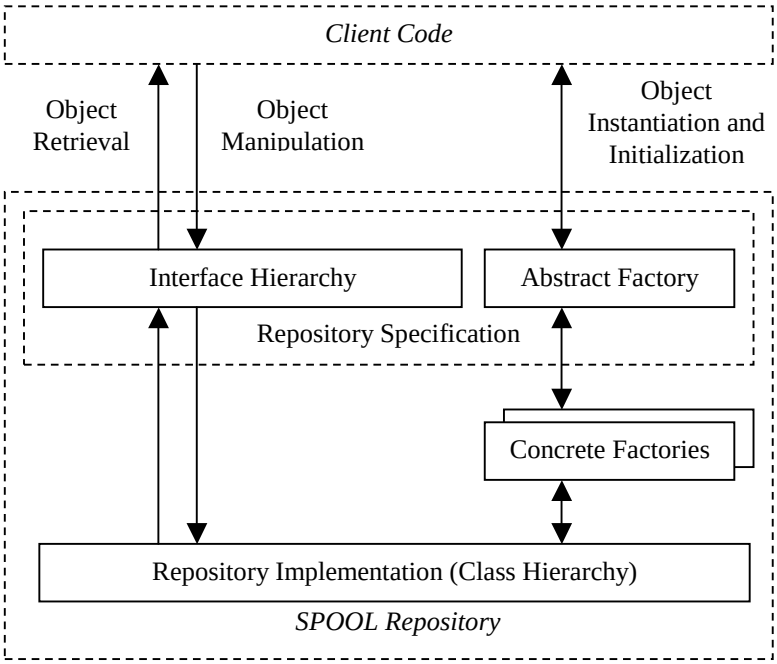

Figure 2. SPOOL repository architecture.

The *interface hierarchy* specifies the semantics of the *retrieval* and *manipulation* functionality of the SPOOL repository. Binding the *client code* to interfaces instead of classes yields the benefit that the client code remains unaware of the concrete types and the implementation of the objects it instantiates. This permits changes in the *repository implementation* without affecting the client code, neither at compile nor at run-time, as long as the implementation adheres to the specification of the interfaces. The *Abstract Factory* provides hook methods for object *instantiation* and *initialization*. Instead of instantiating directly the classes of the repository implementation, the client code requests the instantiation of repository classes from the Abstract Factory class, whose subclasses, the *Concrete Factories*, will perform the actual instantiation of the respective classes. The use of the Abstract Factory design pattern proved very helpful as both the instantiated class and the initialization code can vary among the hook methods of the different Concrete Factories. In this way, the access to the SPOOL repository can easily be adapted, in order to meet application-specific needs.

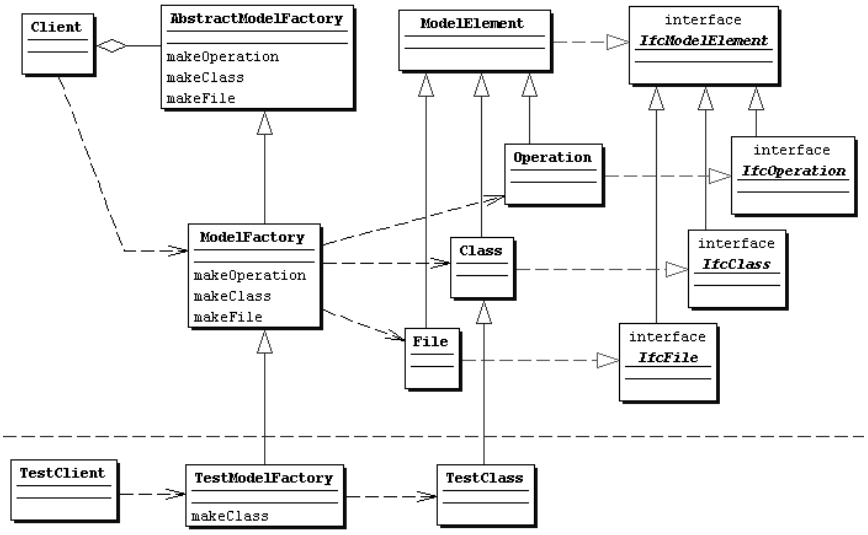

Figure 3. Decoupling of client code from SPOOL repository implementation.

Figure 3 shows an excerpt of the interface hierarchy of SPOOL, which defines *IfcModelElement* to be the parent interface of *IfcOperation*, *IfcClass*, and *IfcFile*. These interfaces are implemented with an abstract *ModelElement* as the superclass and *Operation*, *Class*, and *File* as the respective concrete subclasses. The *Client* code is only bound to the interfaces of the hierarchy (not shown in Figure 3) and to the Abstract Factory *AbstractModelFactory*, which provides the hook methods *makeOperation*, *makeClass*, and *makeFile* for the instantiation of the respective concrete ModelElement classes. The actual instantiation is encoded in the Concrete Factory class *ModelFactory*, in which, for example, *makeClass* returns an instance of the repository class *Class*[1].

This design provides to the SPOOL environment the flexibility needed for corrective changes without any effect on the client code, for simplifying testing, and for the adaptation of the repository to client specific requirements. Due to the decoupling of the client code from the repository's implementation code, corrective changes or refactoring measures in any of the classes that implement the repository will not affect the client code, as long as the changes do not violate the expectations of the client code on the interface. For testing purposes, a new implementation of, for example, *Class* (*TestClass*) could be instantiated in a new subclass of the factory *ModelFactory* (*TestModelFactory*) which inherits all instantiation and initialization code from the established *ModelFactory* and may

[1] Note the difference between the notion of *Class* as used in (a) the repository schema to represent the structure and behavior of the reverse engineered system classes and (b) in the Java programming language as the metaclass of all Java classes.

redefine only *makeClass* to instantiate *TestClass* instead of *Class*. The client code can then be tested easily by just changing the model factory from *ModelFactory* to *TestModelFactory*. A similar solution may be applied in the case in which a specific client of the repository demands functionality that is not implemented in the default version of the repository. Providing the client with a domain-specific ModelFactory can avoid many changes in the client code that would be inevitable if the client code were directly coupled to the repository's implementation.

Z.3 Repository Schema

The schema of the SPOOL repository is an object-oriented class hierarchy whose core structure is adopted from the UML metamodel. Being a metamodel for software analysis and design, the UML provides a well-thought foundation for SPOOL as a design comprehension environment. However, SPOOL reverse engineering starts with source code from which design information should be derived. This necessitates extensions to the UML metamodel in order to cover the programming language level as far as it is relevant for design recovery and analysis. In this section, we present the structure of the extended UML metamodel that serves as the schema of the SPOOL repository. This includes the top-level classes, the core classes, the relationship classes, the behavior classes, and the extension classes.

Z.3.1 Top-Level Classes

The top-level classes of the SPOOL environment prescribe a key architectural design decision, which is based on the *Model/View/Controller (MVC)* paradigm of software engineering (Buschmann et al, 1996; Gamma et al., 1995). MVC suggests a separation of the classes that implement the end user tools (the views) from the classes that define the underlying data (the models). This allows for both views and models to be reused independently. Furthermore, MVC provides for a change notification mechanism based on the *Observer* design pattern (Gamma et al., 1995). The Observer pattern allows tools, be they interactive analysis or background data processing tools, to react spontaneously to the changes of objects that are shared among several tools. In SPOOL, the classes *Element*, *ModelElement*, and *ViewElement* (Figure 4) implement the functionality that breaks the SPOOL environment apart into a class hierarchy for end user tools (subclasses of *ViewElement*) and a class hierarchy for the repository (subclasses of *ModelElement*). The root class *Element* prescribes the MVC based communication mechanism between ViewElements and ModelElements. In the following, we describe these three core classes of SPOOL in more detail.

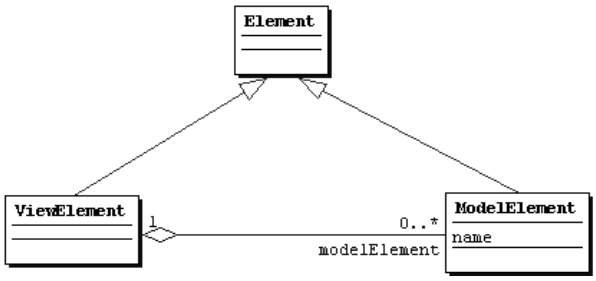

Figure 4. SPOOL repository schema: Top-level classes.

"An *Element* is an atomic constituent of a model" (UML, 1997). It is the abstract superclass from which all SPOOL classes inherit, be they part of the user interface or part of the repository schema. Like other well-known object-oriented framework architectures, such as Smalltalk and *ET++* (Weinand et al., 1989), the top-level *Element* class provides both the *Subject* and the *Observer* interfaces of the MVC architectural design pattern, allowing in principle every SPOOL object to observe any other SPOOL objects. Note that the design and programming guidelines of SPOOL prohibit that ModelElements observe the state of ViewElements. In SPOOL, dependencies emanate always from the ViewElement class hierarchy towards the ModelElement class hierarchy and *never* vice versa.

"A *ModelElement* is an *Element* that is an abstraction drawn from the system being modeled. Contrast with *ViewElement*, which is an Element whose purpose is to provide a presentation of information for human comprehension" (UML, 1997). Each ModelElement has a name, which must be unique in the namespace (see Core Classes) in which it is embedded. The class hierarchy of ModelElement represents the SPOOL repository schema. It comprises classes that represent the structure and behavior of both the reverse engineered source code models and the higher-level abstractions of the systems that are recovered by the visualization and analysis tools.

"A *ViewElement* is a textual and/or graphical projection of a collection of *ModelElements*" (UML, 1997). The ViewElement class is the abstract root class of the SPOOL tools, which provides the abstract functionality and specifications for rendering of user interface objects, be they complex diagrams or primitive graphic objects. A ViewElement holds on to the ModelElements that provide the data for the visual or textual representation. Upon notification of a change in ModelElements, implemented via the observation mechanism in Element, the ViewElement fetches the relevant data from the respective ModelElements and redraws the relevant parts of its representation.

Z.3.2 Core Classes

The core classes of the SPOOL repository schema adhere to a large extent to the classes defined in the core and model management packages of the UML metamodel. These classes define the basic structure and the containment hierarchy of the ModelElements managed in the repository (see Figure 5). At the center of the core classes is the *Namespace* class, which owns a collection of ModelElements. A *GeneralizableElement* defines the nodes involved in a generalization relationship, such as inheritance. A *Classifier* provides *Features*, which may be structural (*Attributes*) or behavioral (*Operations* and *Methods*) in nature (see Figure 6). A *Package* is a means of clustering ModelElements. In the following, we will detail the Namespace and the Feature classes.

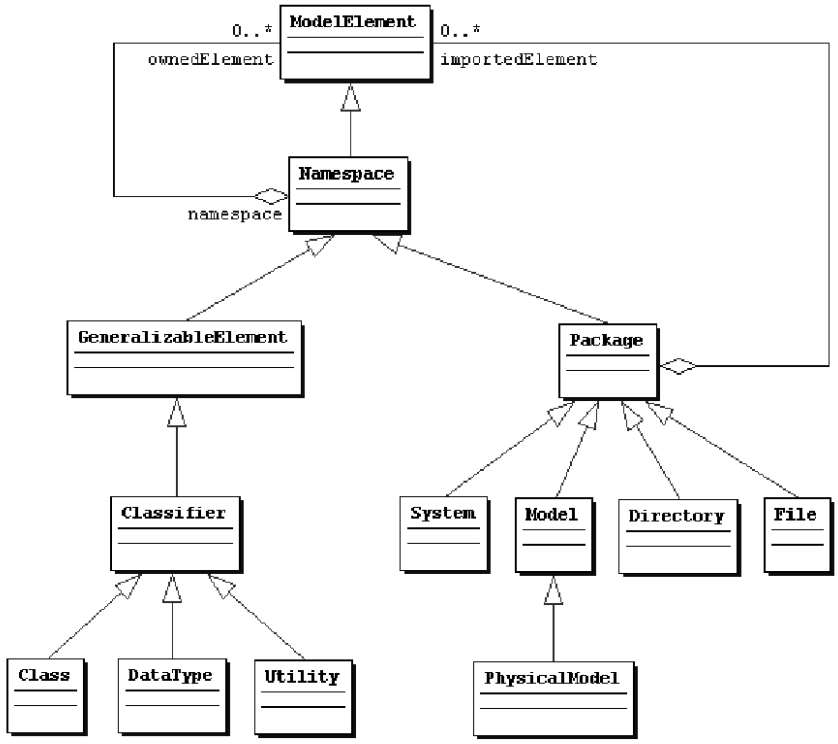

Figure 5. SPOOL repository schema: Namespace classes.

"A *Namespace* is a part of a model that contains a set of ModelElements each of whose names designates a unique element within the namespace" (UML, 1997). A Namespace may be viewed as a container for ModelElements, which

themselves may be Namespaces. The Namespace defines an owning (or existence) relationship to these ModelElements, meaning that the contained ModelElements cease to exist if the containing Namespace is deleted from the repository. Examples of Namespaces are classes, unions, files, directories, or whole systems. In SPOOL, the Namespace provides much functionality for retrieval and traversal of complex source code structures, such as directories, which may contain other directories or source code files, which in turn contain classes, which are the containers for methods and attributes, and so forth. As an example of such a traversal routine, the Namespace method *allContainedElements (Method.class)* applied to a whole reverse engineered system returns all methods that are somewhere contained in any class stored in any file of any subdirectory of the system at hand.

"A *GeneralizableElement* is a ModelElement that may participate in a generalization relationship" (UML, 1997). GeneralizableElements are the end points of a Generalization relationship. Refer to the Relationship classes described below for further details on generalization hierarchies. Subclasses of GeneralizableElement are Classifier, an abstract superclass for classes, unions, interface, and alike, and Package, which may be a whole system, a directory, or a file.

"A *Classifier* is an element that describes behavioral and structural features; it comes in several specific forms, including class, data type, interface, component, and others that are defined in other metamodel packages" (UML, 1997). In SPOOL, the Classifier implements all access mechanisms to its features, which are attributes, operations, and methods. Being a subclass of Namespace, Classifier inherits the storage, retrieval, and manipulation functionality for its features from Namespace. Moreover, it allows for nested Classifiers, such as the inner or anonymous classes of Java. A key role in the SPOOL repository plays the *Utility* subclass of Classifier. It serves as a container for all non-member behavioral and structural features, which are declared or defined outside the namespace of a class. Again, being a subclass of Classifier, Utility reuses all repository functionality for the management of such global features.

"A *Package* is a grouping of model elements" (UML, 1997). In SPOOL, Package is the superclass for containers of ModelElements. For example, a typical containment hierarchy of a reverse engineered model starts with a system, which contains physical and logical models. A *PhysicalModel* consists of *Directories* or *Files*, where the Directories may contain other Directories or Files. Files may be composed of Classes, Unions, Datatypes, a Utility for the global code, and alike. The traversal functionality for such complex package structures is inherited from Namespace, which allows querying direct and indirect package containment.

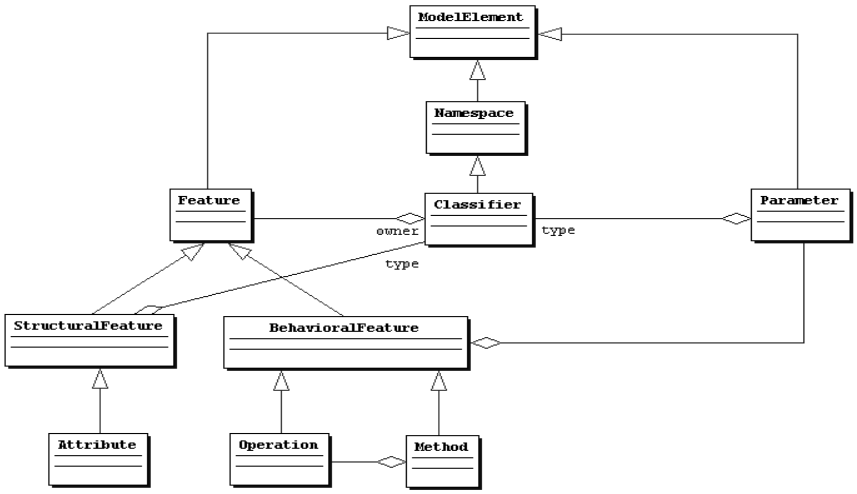

Figure 6. SPOOL repository schema: Feature classes.

"A *Feature* is a property, like operation or attribute, which is encapsulated within a Classifier" (UML, 1997). A feature can be structural or behavioral in nature. *StructuralFeatures* are *Attributes* whose type is a Classifier. *BehavioralFeatures* can be *Operations* or *Methods*, and they are associated with a set of *Parameters*, which also include the return parameter. In object-oriented literature there is much confusion about the difference between Operations and Methods and, very often, these two notions are used synonymously. However, there is a significant difference between these two notions. The UML defines an Operation as a service that can be requested from an object of a class and a Method as the implementation of an Operation. Whereas an Operation constitutes a specification, a Method provides the executable body of the algorithm implementing the specification. Every Method must have exactly one Operation, which can be inherited from supertypes in the generalization hierarchy. An Operation can be implemented multiple times within its subtype hierarchy.

Z.3.3 Relationship Classes

"A relationship is a connection among model elements" (UML, 1997). The UML introduces the notion of Relationship as a superclass of Generalization, Dependency, Flow, and Association for reasons of convenience, so that tools can

refer to any connections among ModelElements based on the same supertype (Figure 7).

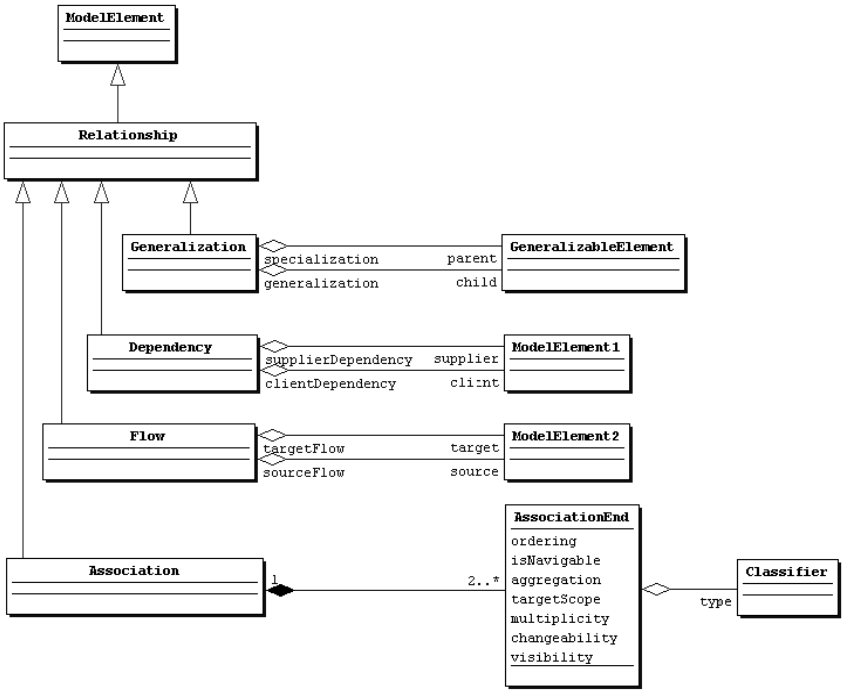

Figure 7: SPOOL repository schema: Relationship classes.

"A *Generalization* is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information" (UML, 1997). In SPOOL, classes, interfaces, and packages can participate in a generalization relationship, since they all are GeneralizableElements (see section Core Classes). It is common knowledge that classes and interfaces can be organized in form of a generalization hierarchy; however, it is less obvious that packages can participate in a generalization hierarchy, too. The purpose of organizing packages in form of a generalization hierarchy is to show design solutions at multiple levels of abstraction. In SPOOL we use package generalization to structure design patterns (Keller et al., 1999). For example, the *Composite* design pattern may be implemented as a *Safe Composite* or a *Transparent Composite*, each having its strength and pitfalls. Refer to Gamma et al. (1995) for more details. At a certain level of de-

sign comprehension, however, these differences can or should be ignored, to be able to keep a global overview of the system at hand and not get bogged down into design details.

"A *Dependency* states that the implementation of functioning of one or more elements requires the presence of one or more other elements" (UML, 1997). The UML specifies the modeling of Dependencies as a *client*/*supplier* relationship between ModelElements and associates to each of the two participating ModelElements the Dependency that serves as the *supplier* (*supplierDependency*) and the one that serves as the *client* (*clientDependency*), respectively. Examples of dependencies as stated by the UML are Bindings, Abstractions, Usage, or Permissions (UML, 1997). We implemented the UML dependency mechanism to allow SPOOL tools to analyze and store semantic relationships among the model elements in the repository. Conforming to the UML metamodel, we also use dependencies to store some structural relationships between model elements, such as friendship relationships among classes of a reverse engineered C++ system.

"A *Flow* is a relationship between two versions of an object or between an object and a copy of it" (UML, 1997). A flow is directed, it emanates from a source towards a target. In the SPOOL repository, we implement the Flow relationship to provide a foundation for version management for reverse engineered source code models. The objective is to store only the delta that has changed between two versions of the system at hand.

"An *Association* defines a semantic relationship between classifiers. The instances of an association are a set of tuples relating instances of the classifiers. Each tuple value may appear at most once" (1997). The ends of an Association are called *AssociationEnd*, which carry the semantics of the relationship to the classifier at the respective end. This includes information on ordering (objects at the end of an association can be organized in ordered, sorted, unordered, or other data structures), navigability (possibility of traversal from one end to the other), aggregation (designation of part/whole relationships), target scope (specification if the relationship points to a class or an instance), multiplicity (number of instances of the associated classifier that can participate in the association), changeability (specification if the classifier at one end can be modified from the one of the other end), and visibility (specifies visibility of the classifier at association end to the classifier at the opposite end). Refer to the UML (UML, 1997) for more details on the semantics of these attributes. Note that the derivation of Associations and the attributes of the AssociationEnds is not a straightforward task. The goal of the SPOOL reverse engineering environment is to provide human-controlled tool support to this end.

Z.3.4 Behavior Classes

The behavior classes of the SPOOL repository implement the dynamics of the reverse engineered system. It is important to understand that the UML meta-

model takes a forward engineering perspective and focuses on software analysis and design, rather than on the reverse engineering of source code. Therefore, the UML metamodel does not aim to encompass and unify programming language constructs.

The purpose of analysis and design is to specify what to do and how to do it, but it is the later stage of implementation in which the missing parts of a specification are filled to transform it into an executable system. However, the UML is comprehensive in that it provides a semantic foundation for the modeling of any specifics of a model. For example, the UML suggests State Machine diagrams (similar to Harel's Statechart formalism (Harel, 1988)) to specify the behavior of complex methods, operations, or classes. To cite another example, collaboration diagrams can be used to specify how different classes or certain parts of classes (that is, roles) have to interact with each other in order to solve a problem that transcends the boundaries of single classes.

In SPOOL, we look at a system from the opposite viewpoint, that is from the complete source code, and the goal is to derive these behavior specification models to get an improved understanding of the complex relationships among a system's constituents. For this purpose, we included in the SPOOL repository the key constructs of the behavior package of the UML metamodel, including the Action and Collaboration classes presented below. However, we modified certain parts to reduce space consumption and improve performance.

Action Classes

"An *Action* is an executable atomic computation that results in a change in state of the system or the return of a value" (Booch et al., 1999). The SPOOL repository uses Actions to describe the internals of a method. Figure 8 shows the corresponding diagram.

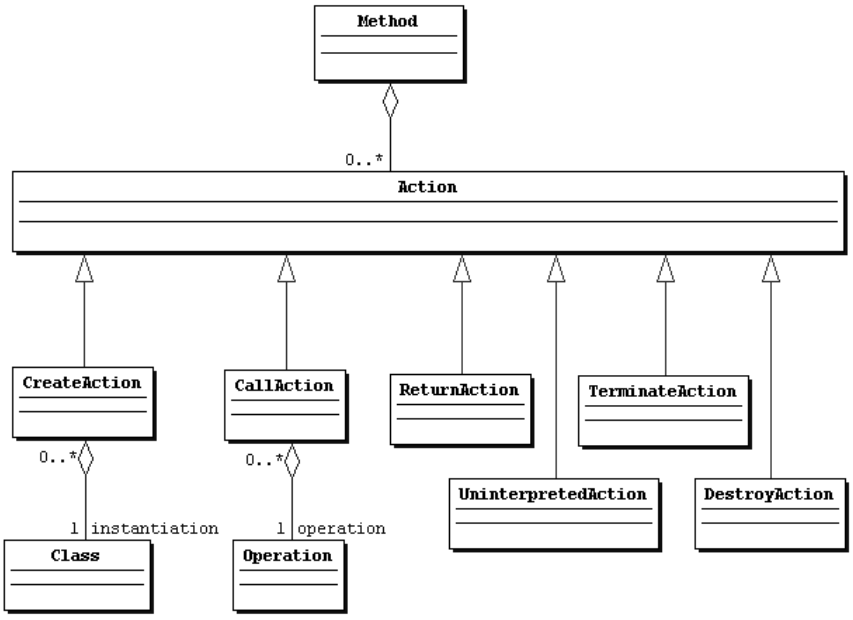

Figure 8. SPOOL repository schema: Action classes.

The UML metamodel embeds Actions into State Machines and Collaborations, which would result in an extra StateMachine object for each method of the reverse engineered system. In SPOOL, we provide a shortcut and save the set of Actions directly with the Method and provide access as well as manipulation routines for Actions as part of the SPOOL schema class Method. This improves performance of many queries on the SPOOL repository as it avoids indirection via StateMachine objects. Furthermore, the UML specifies many different subclasses of Action, which include *CreateAction*, *CallAction*, *ReturnAction*, *TerminateAction*, *UninterpretedAction*, and *DestroyAction* (UML, 1997). We implemented all of these classes, but as the SPOOL analysis tools that we implemented so far are based on CreateActions and CallActions only, we do not import all this information into the SPOOL repository. This reduces the overall size of the physical datastore, which in turn improves the performance of many analysis tools. Note that at any time objects instantiating any of the Action subclasses can be imported, as the integrated Datrix parsers (Datrix, 2000) generate all the necessary information. Such an extension would not affect the existing SPOOL visualization and analysis tools.

Collaboration Classes

One of the key goals of the SPOOL reverse engineering environment is to provide support for the extraction of predefined design concepts, such as the

structures of design patterns, from source code. We call the design concepts to be recovered abstract design components and the recovered instances of these design concepts implemented design components. The notion of design component alludes to the fact that these conceptual fragments of the overall design are managed as entities that may be used as the building blocks in a compositional design process (Keller and Schauer, 1998). In SPOOL, we implemented a simplified version of the Collaboration package defined by the UML metamodel (Figure 9).

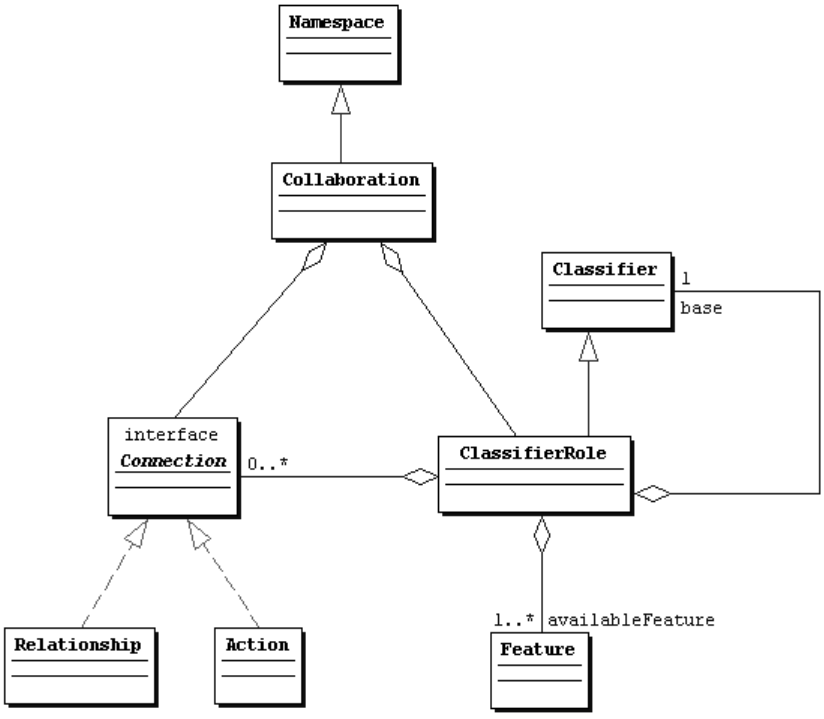

Figure 9: SPOOL repository schema: Collaboration classes.

"A *Collaboration* describes how an operation or a classifier is realized by a set of classifiers and associations used in a specific way" (UML, 1997). In the UML metamodel, a Collaboration can be viewed as a set of interacting roles, which define the communication among classes to achieve some overall functionality.

"A *ClassifierRole* is a specific role played by a participant in a Collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration" (UML, 1997). A ClassifierRole is defined by the set of *available Features* of the *base* Classifier, which altogether play a certain role in the Collaboration at hand.

In the UML metamodel, ClassifierRoles are related by AssociationRoles, which are roles on Associations. In SPOOL, this mechanism is not needed, since we are dealing with physical connections in the source code, such as instantiations and function calls. Therefore, we modified the design by introducing an interface *Connection* as a supertype of all subclasses of ModelElement that represent links among ModelElements. This includes *Relationship* and *Action*, as well as all their subclasses. The Connections can be associated with the ClassifierRoles of Collaboration. Note that the Connections of a ClassifierRole can only be a subset of all the Connections that are defined by the Classifier, which is the base of the ClassifierRole.

Z.3.5 Extension Classes

The UML metamodel suggests two approaches to metamodel extension; one is based on the concept of TaggedValues and the other on the concept of Stereotypes. In SPOOL, we have only implemented the former approach since Stereotypes as defined in the UML metamodel would not scale to meet the performance requirements of the SPOOL repository.

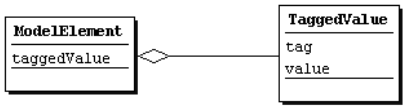

Figure 10. SPOOL repository schema: TaggedValue extension class.

"A *TaggedValue* is a (Tag, Value) pair that permits arbitrary information to be attached to any ModelElement" (UML, 1997). In programming languages, the concept of TaggedValue is known under the notion of property. Using TaggedValues, the end user tool can plug into any ModelElement any tool-specific data. This may include information on color, layout, paths, aliases, and the like. The interpretation of the TaggedValue is up to the end user tool, and different tools must make sure that they use the TaggedValues consistently.

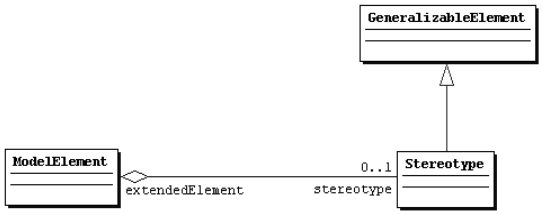

Figure 11. SPOOL repository schema: Stereotype extension class.

"The *Stereotype* concept provides a way of classifying (marking) elements so that they behave in some respects as if they were instances of new *virtual* metamodel constructs" (UML, 1997). As a concrete example, the UML metamodel suggests to model the Utility (classifiers for non-member features) as a stereotyped *Class*. In such a setting, Utility objects would be instances of the UML metaclass Class and associated with the Stereotype object of the name Utility. Applied to reverse engineering, a C++ file with a class definition and some global variable definitions would be mapped in the repository as an instance of the metaclass File containing two instances of the metaclass Class, one for the class definition and the other for the global variable definition. The latter would be marked with the *Utility* stereotype. The pitfall of such a design for the SPOOL repository is that traversal methods would always need to verify if an object is a pure instance of the metaclass Class or a stereotyped instance. This would result in unacceptable performance when traversing a system with thousands of classes. Note that, to make things even worse, the UML metamodel, unlike the SPOOL schema, (see Figure 5) defines the File class as a stereotype of Package. This is the reason why the SPOOL repository refrains from using Stereotypes as defined by the UML metamodel. Rather, it represents all extensions to the basic metamodel, such as Utility or File, as subclasses of the metaclass Namespace. Thus, each ModelElement has an unambiguous type of its metaclass, and the Namespace traversal methods can use the Java *instanceof* operator to identify the type of a ModelElement.

Z.4 Repository Mechanisms

To be usable and reusable as the backend for a diverse set of interactive reverse engineering tools, the SPOOL repository implements a number of advanced mechanisms. The *traversal mechanism* defines how to retrieve objects of certain types from a complex object containment hierarchy. *The observation mechanism* defines model/view change notification that goes beyond the Observer pattern. The *dependency mechanism* allows for compression of the vast amount of dependencies among ModelElements for fast retrieval and visualization.

Z.4.1 Traversal Mechanism

In SPOOL, the Namespace serves as a container for a group of ModelElements (Figure 5). Consequently, it defines methods that traverse complex object structures and retrieve ModelElements of a given type. For example, to identify all classes of a system, all files in all subdirectories of the system at hand must be checked for instances of the metatype Class. Unlike the objects in text-based repositories (Wong and Müller, 1998; Holt, 1997), the objects in SPOOL's ob-

ject-oriented database are typed and can be queried according to their types. SPOOL allows for the identification of the type of an object merely by using the Java *instanceof* operator or the reflective *isInstance* operation of the Java class *Class*. Hence, metaclass types can be provided as parameters to the retrieval methods of Namespace, which then recursively traverse the containment hierarchy of the namespace at hand and examine each ModelElement whether it is an instance of that type. If this is the case, the ModelElement is added to a return set, which is passed through the recursive traversal. The following code snippet shows the implementation of the method *allContainedElements* of the SPOOL repository class Namespace.

```
public void allContainedElements(Class aMetaClass, Set returnSet) {
    // elements() returns an iterator over the direct content
    // of a namespace
    Enumeration enum = elements();
    while (enum.hasMoreElements()) {
        ModelElement modelElem = (ModelElement) enum.nextElement();
        if (aMetaClass.isInstance(modelElem))
            returnSet.add(modelElement);
        if (modelElem instanceof Namespace)
            ((Namespace) modelElem).
                   allContainedElements(aMetaClass, returnSet);
    }
}
```

The above version of *allContainedElements* accepts two parameters, the first specifies the type of the instances of ModelElement to be retrieved and the second accepts a Set in which the retrieved ModelElements are to be returned. For each ModelElement of the Namespace to which *allContainedElements* is applied, this method first verifies if the ModelElement at hand is an instance of the type passed as parameter; if so, it is added to the return set. Then, it checks if the ModelElement is itself an instance of a Namespace, in which case *allContainedElements* is recursively applied to this sub-Namespace. Recall that the SPOOL repository schema is an object-oriented class hierarchy and, therefore, traversal and retrieval operations can also use abstract superclasses or interfaces to identify ModelElements of any of the derived subclasses. This is very helpful for SPOOL tools to query the content of the repository. In many cases, there is no need to change the client code if the repository schema is, for example, extended with a new leaf class. Client code that is bound to the interface of an abstract superclass of the SPOOL repository schema will automatically receive objects of the new subclass.

Z.4.2 Observation Mechanism

A general goal of the design of the SPOOL repository is to serve as the back-end for interactive reverse engineering tools. Multiple end user tools should be able to work in parallel without sacrificing data consistency. This calls for a change notification mechanism that informs running tools about manipulations in the repository. Hence, SPOOL is based on the Model/View/Controller architectural design pattern (Buschmann et al., 1996), which separates the user interfaces (the ViewElement class hierarchy) from the application data (the ModelElement class hierarchy) and allows for the synchronization of multiple user interfaces on the same data.

Applying MVC out of the textbook to the vast amount of data that is typically stored in the SPOOL repository would soon degenerate overall performance due to considerable runtime overhead for model/view coordination (Vlissides, 1998). For example, ET++ (Weinand et al., 1989), a small-sized C++ application framework for graphic user interfaces of about 70,000 lines of C++ source code contains about 600 classes with 7,000 methods. If we loaded all of these 600 classes with all 7000 methods into a class diagram, and applied the pure Observer design pattern to coordinate the views with possible model changes, this would result into 7,600 graphic representations each hooked into their model counterparts upon creation of a system's class diagram. Vice versa, when the class diagram is closed, these views would need to be removed again from their respective models. As another example, consider source code clustering tools, which usually shift many classes and other model elements around in the system to be re-organized. Their performance would greatly suffer from a model that generated thousands of change notification messages for each removal of a ModelElement from one Namespace and its subsequent addition to another Namespace. In theory, the SPOOL repository would allow such a strategy, but in practice this is only feasible for small-sized diagrams, where message passing does not lead to a performance bottleneck.

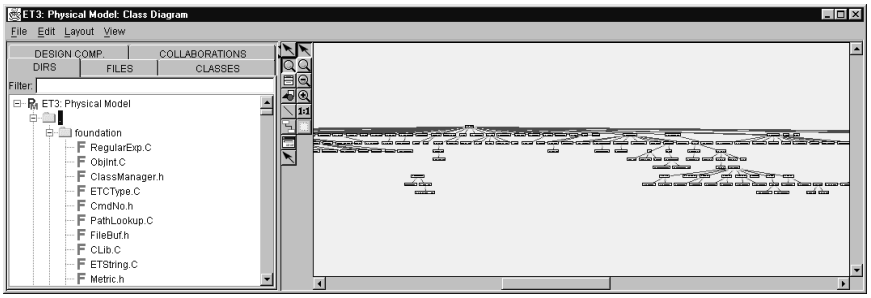

Figure 12. View synchronization via SPOOL's observation mechanism.

In SPOOL, we implemented another strategy for model/view change notification. Diagrams as a whole, such as the tree directory diagram (left part of Figure 12) and the class diagram (right part of Figure 12), are hooked into the namespaces that they represent, which is in the given case the whole system ET++ (Weinand et al., 1989). When a ModelElement of any part of the system is updated or a ModelElement is added or removed, the SPOOL repository propagates notification of the change to all containers of this ModelElement. If a view observes any of these containers it will be notified of the change. In Figure 12, for instance, if a method is deleted from any class, the SPOOL repository generates a change event indicating this update, and propagates it along the container path, which includes the class, the file of the class, all directories in which this file is contained, the physical model that holds the reverse engineered code, and finally the system as the outermost container. As the tree directory diagram and the class diagram are hooked into the system, they receive this event, which includes information about what and where the event occurred. As a result, each diagram can identify the changed element in its containment hierarchy and execute the appropriate update on its visual representation. This avoids the problem of too many runtime links between models and observers; however, it does not address the problem that every change results in an event that is separately passed on to the observing views.

The manipulation of ModelElements in the SPOOL repository is performed based on transactions. To guarantee consistency among end user tools, the SPOOL repository caches all change events in what SPOOL calls aggregate event sets. These sets aggregate all changes of the SPOOL repository within a transaction. When the transaction is finished, all view elements that hook into one of the changed model elements or any of its containers are notified of the change and receive the set of changed model elements they are observing. Normally, such aggregate event sets can significantly reduce message passing between models and views.

Z.4.3 Accumulated Dependency Mechanism

An important requirement of the SPOOL repository is to provide information on dependencies between any pair of ModelElements within interactive response time. Figure 13 illustrates this requirement in more detail.

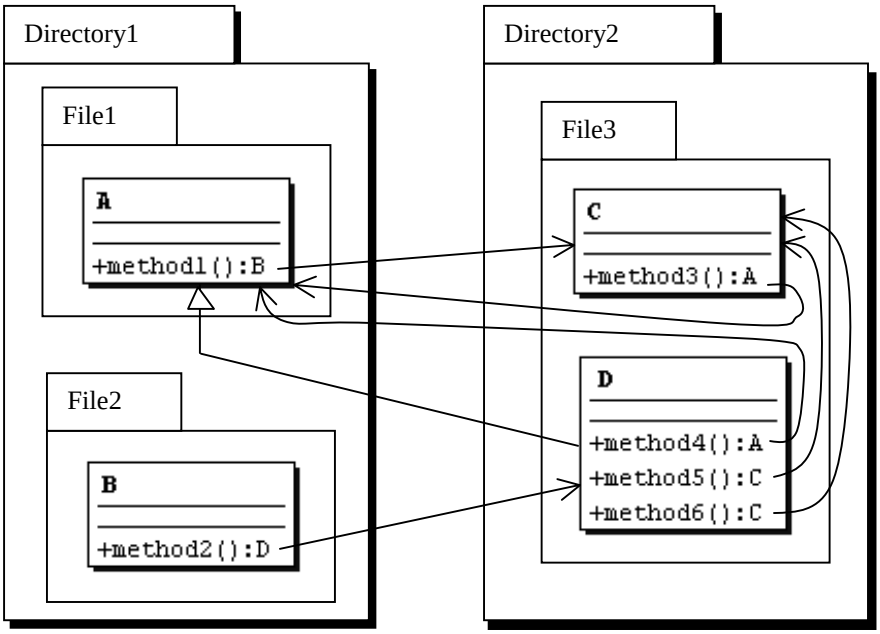

Figure 13: Dependencies between ModelElements.

The directory *Directory1* of Figure 13 consists of the two files *File1*, which contains the class *A*, and *File2*, which contains the class *B*. Directory *Directory2* consists of the file *File3*, which contains the two classes *C* and *D*, where *D* is a subclass of *A*. The return type of the method *method1* is *C*, the return type of the method *method2* is *D*, the return type of the methods *method3* and *method4* is *A*, and the return type of the methods *method5* and *method6* is *C*. In this diagram, there are seven connections among ModelElements. Six connections are defined by the return types (*ReturnType* connection) of the methods, and one connection is defined for the generalization relationship (*Generalization* connection) between *A* and *D*. According to the above mentioned requirement, end user tools should be able to identify, within interactive response time, whether there are any connections between, say, the two top-level directories *Directory1* and *Directory2*, the directory *Directory1* and the method *method3* of class *C*, or the file *File2* and the class *D*, and display them in dependency diagrams. Figure 14, for example, shows such a dependency diagram for the top-level directories of the system ET++ (Weinand et al., 1989). A property dialog box can be opened to inspect the nature of a specific dependency. In Figure 14, for instance, the dependency between the directories *CONTAINER* and *foundation* includes 13 generalization connections, 50 feature type connections (types of attributes and return types of operations and methods), 541 parameter type connections, 5 class instantiation connections (*CreateAction*), 498 operation call connections (*Cal-*

lAction), and 0 friendship connections. On demand, the dialog can also be invoked for each direction of a dependency.

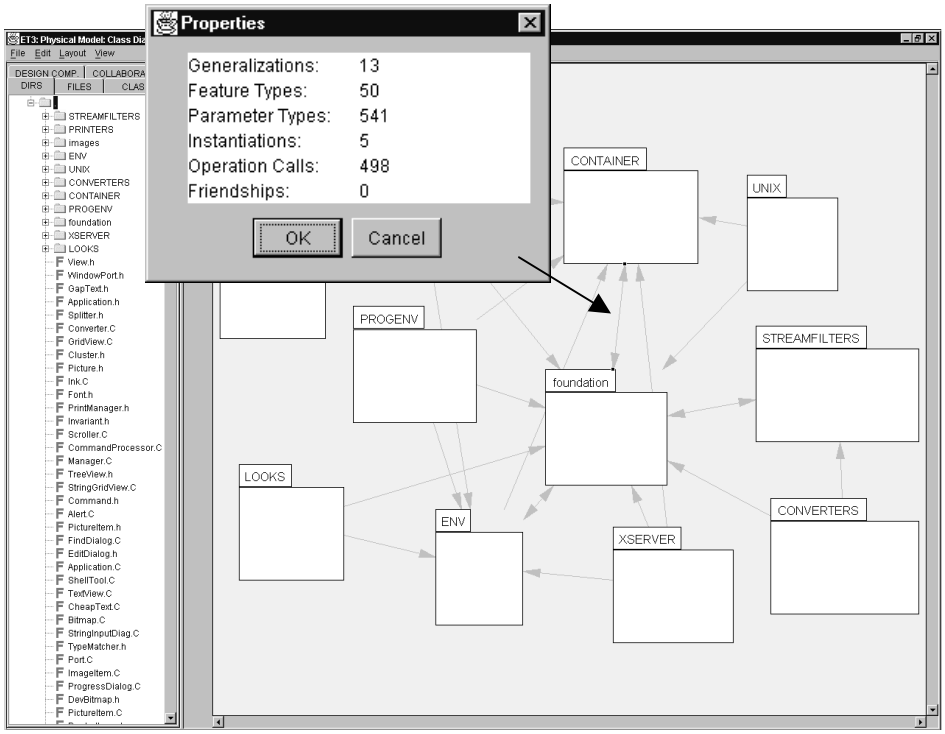

Figure 14. SPOOL dependency diagram with dialog box for inspection of properties.

A straightforward approach to identify dependencies among ModelElements would be the traversal of the whole object structure at run-time. However, applied to reverse engineered software with directories that contain hundreds of files, this approach would require batch processing. A radically different approach would be to store each and every dependency among ModelElements as separate dependency objects, which would result in an unmanageable amount of dependency data. In the small example of Figure 13, this would amount to an overall of 36 dependency objects. Hence, the solution that we adopted in SPOOL constitutes a trade-off between run-time efficiency and space consumption.

In SPOOL, we capture and accumulate dependencies at the level of Classifiers (for instance, classes, unions, or utilities). Accumulation refers to the fact that

we store for each dependency its types together with the total number of primitive Connections on which each type is based. Given a pair of dependent Classifiers, we generate a so-called *AccumulatedDependency* object, which captures this information for the dependencies in the two directions. To be able to identify dependencies between higher-level namespaces, such as directories, files, or packages, without much lag time, we store the union of all AccumulatedDependencies of all contained Classifiers of a given Namespace redundantly with the Namespace. Hence, if we want to identify, for example, dependencies between the directories *Directory1* and *Directory2*, we only need to iterate over the set of AccumulatedDependencies of *Directory1* and look up for each element of the set whether the ModelElement at the other end of the element at hand (that is, the one which is not contained in the Namespace under consideration) has as one of its parent namespaces *Directory2*.

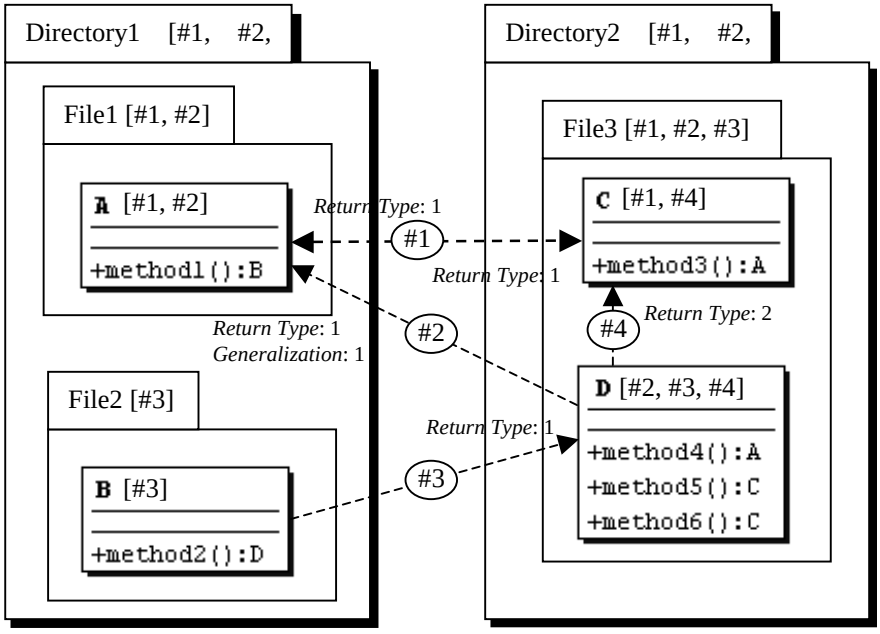

Figure 15: AccumulatedDependencies between ModelElements.

Figure 15 illustrates the *AccumulatedDependencies* of our previous example. *A* includes the *AccumulatedDependencies* #1 and #2, *B* the *AccumulatedDependency* #3, *C* the *AccumulatedDependencies* #1 and #4, and *D* the *AccumulatedDependencies* #2 and #3. Each *AccumulatedDependency* is designated with the types and the number of connections on which it is based, in either direction. The *AccumulatedDependency* #1 is based on one *ReturnType* connection from *A* towards *C* and one in the opposite direction from *C* to *A*, the *AccumulatedDependency* #2 is based on one *ReturnType* connection and one *Generalization*

connection in the direction of *D* towards *A*, the *AccumulatedDependency* #3 is based on one *ReturnType* connection from *B* towards *D*, and the *Accumulated-Dependency* #4 is based on two type *ReturnType* connections emanating from *D* towards *C*. All *Namespaces* collect all *AccumulatedDependencies* of their contained *Namespaces*, which amounts, for instance, for the directories *Directory1* and *Directory2* to the *AccumulatedDependencies* #1, #2, and #3, for *File1* to the *AccumulatedDependencies* #1 and #2, for *File2* to the *AccumulatedDependency* #3, and for *File3* to the *AccumulatedDependencies* #1, #2, and #3.

Z.5 Conclusion

The authors of the UML, Booch, Jacobson, and Rumbaugh, acknowledge that "reverse engineering is hard; it's easy to get too much information from simple reverse engineering, and so the hard part is being clever about what details to keep" (Booch et al., 1999). Reverse engineering is the human controlled process of transforming the flood of detail information contained in source code into structural and behavioral models at higher levels of abstraction. These are meant for easy comprehension of complex system interrelations, and the UML with its nine different kinds of diagrams was designed to this end. The authors of the UML emphasize that the purpose of the UML is not to provide a mere notation for forward engineering; rather, the UML was devised both "to allow models to be transformed into code and to allow code to be re-engineered back into models" (Booch et al., 1999).

The UML is hardly accepted in the reverse engineering community. Demeyer et al. have articulated some reasons for the "why not UML" (Demeyer et al., 1999). We wholeheartedly agree that there is a lack of complete and precise mappings of programming languages to the UML. However, we consider this as a challenge for researchers, rather than a reason for abandoning the UML. With its Stereotype extension mechanism, the UML does provide constructs to capture the many details of source code written in different programming languages. The issue at hand is to define unambiguously how to map the various UML constructs to source code constructs and to provide tool support for the traceability in both directions. A second argument of Demeyer et al. against the UML is that it "does not include dependencies, such as invocations and accesses" (Demeyer et al., 1999). This constitutes yet another misconception in the reverse engineering community about the UML. All too often, the UML is looked at as a notation for structure diagrams only, and all other diagrams are rather neglected. Yet, the behavior package of the UML metamodel provides a precise specification of the method internals. However, a critique against the UML may be that the behavioral package is too heavyweight to be directly applicable to reverse engineering. It is impossible to generate and store for each method a StateMachine object together with all its internal objects. In SPOOL, we implemented a shortcut solution for the representation of the bulk of the methods. We associated Actions

directly to methods instead of generating StateMachines, which consist of Actions that are invoked by Messages. Refer to the UML for further details on the structure of StateMachines (Booch et al., 1999). We do, however, allow StateMachines to be reverse engineered and stored for methods or classes of interest.

The UML has several advantages. First, the UML metamodel is well documented and based on well-established terminology. This is of great help to convey the semantics of the different modeling constructs to tool developers. Second, the metamodel is designed for the domain of software design and analysis, which is the target of the reverse engineering process. The UML introduces constructs at a high level of granularity, enabling the compression of the overwhelming amount of information that makes source code difficult to understand. Third, the UML metamodel is object-oriented, meaning that the structure, the basic access and manipulation behavior, and complex mechanisms can be separated from end user tools and encapsulated in the repository schema. Fourth, the UML defines a notation for the metamodel constructs, thus providing reverse engineering tool builders guidelines for the visual representation of the model elements.

Adopting the UML in the SPOOL environment has proven to be one of the most important and beneficial decisions of our project. In this chapter, we have described how the UML can be matched to reverse engineering, and what benefits can be reaped. We are not aware of any other implementation of the UML metamodel for reverse engineering purposes; the SPOOL repository constitutes a proof-of-concept of such an implementation.

In our future work on the SPOOL design repository, we will aim to provide complete and precise mappings between the constructs of the UML-based SPOOL repository schema and the four programming languages C, C++, Java, and a proprietary language deployed by Bell Canada. We will also increase the information content of the SPOOL repository in respect to dynamic behavior. As discussed previously, a balance between space consumption and fast response time needs to be sought. One solution that we will investigate is parsing the source code of methods on the fly when querying, for example, control flow information. A third area of work will be to provide Web-based access to the repository, which will allow our project partners to remotely check in source code systems and immediately use SPOOL tools to query and visualize the repository content.

Z.6 References

Booch, B., Jacobson, I., and Rumbaugh, J. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

Buschmann, F. et al. (1996). *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons.

Chikofsky, E. J., and Cross, J. H. II (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13-17, January 1990.

Datrix (2000). Datrix homepage. Bell Canada. On-line at <http://www.iro.umontreal.ca/labs/gelo/datrix/>.

Demeyer, S., Ducasse, S., and Tichelaar, S. (1999). Why unified is not universal. UML shortcomings for coping with round-trip engineering. In *Bernhard Rumpe, editor, Proceedings UML'99 (The Second International Conference on the Unified Modeling Language)*. Springer-Verlag, 1999. LNCS 1723.

Fowler, M. (1997). *Analysis Patterns. Reusable Object Models*. Addison-Wesley.

Gamma E. et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Harel, D. (1988). On visual formalisms. *Communications of the ACM*, 31(5):514-530, May 1988.

Holt, R. C. (1997). Software Bookshelf: Overview and construction. March 1997. On-line at <http://www-turing.cs.toronto.edu/pbs/papers/>.

Keller, R. K., and Schauer, R.. (1998). Design components: towards software composition at the design level. In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, pages 302-310, April 1998.

Keller, R. K. et al. (1999). Pattern-based reverse engineering of design components. In *Proceedings. of the 21st International Conference on Software Engineering*, Los Angeles, pages 226-235, May 1999.

ODMG (2000). Object Data Management Group (ODMG). On-line at <http://www.odmg.com>.

POET (2000). Poet Java ODMG binding, on-line documentation. Poet Software Corporation. San Mateo, CA. On-line at <http://www.poet.com>.

Tsichritzis, D. and Klug, A. C. (1978). The ANSI/X3/SPARC DBMS framework report of the study group on dababase management systems. *Information Systems*, 3(3):173--191, Pergamon Press Ltd.

UML (1997). Documentation set version 1.1. On-line at <http://www.rational.com>.

Vlissides, J. (1998). *Pattern Hatching. Design Patterns Applied*. Addison-Wesley, 1998.

Weinand, A., Gamma, A., and Marty, R.. (1989). Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63-87, February 1989.

Wong, K. and Müller, H. (1998). Rigi user's manual, version 5.4.4, University of Victoria, Victoria, Canada. On-line at <ftp://ftp.rigi.csc.uvic.ca/pub/>.