Probe: Visualizing Algebraic Transformations in the GenBorg Generative Software Environment

Diplomarbeit[1]

Axel Rauschmayer
Institut für Informatik
Ludwig-Maximilians-Universität München
rauschma@informatik.uni-muenchen.de

December 30, 2001

[1]Thanks to Prof. Don Batory (Department of Computer Sciences, University of Texas at Austin) for supervising and supporting this thesis.

Abstract: Software is becoming increasingly pervasive and complex. The greatest challenge for the industry today is how to produce more code without compromising quality or cost. *GenBorg* is a programming paradigm whose answer is to scale the units of reuse from components of individual applications to components of *application suites* (e. g., MS Office). While traditional approaches trade size of code units for flexibility, GenBorg uses techniques from generative programming and feature-oriented programming in order to provide *models* that instantiate code and non-code artifacts of customizable application suites. Because developing software is not limited to writing code, but current ways of automating development are, a lot of human energy is wasted for keeping related data such as documentation and formal properties consistent with source code. GenBorg manages to evolve all these artifacts automatically and in parallel. GenBorg's ideas are backed by a theoretical foundation, the GenBorg algebra. The Java application Probe is a first implementation of the GenBorg programming paradigm and uses a simple, file-based data structure for defining GenBorg models. Its graphical user interface makes it easy to navigate through and instantiate from large models.

Contents

Chapter 1

Introduction

Software is becoming increasingly pervasive and complex (table 1). The greatest challenge for the software industry today is how to produce more code without compromising quality or cost. One way of achieving this—and the main driving force behind the industry's adoption of OOP[1]—is to improve code reuse. Most approaches for improving reuse have focussed on *vertical scaling*, the idea that if one increases the size of the building blocks in, for example, a library, more code is reused each time one of these blocks is deployed. Accordingly, OOP's units of reuse have been scaled up from classes to components to frameworks.

Alas, the increased size of a block of code (which we will henceforth call a component, regardless of its size) usually means that it becomes more specialized and the number of potential applications decreases: The performance of a component becomes unacceptable, because it cannot be optimized for a specific task. Adding new features that have not been considered when the component was initially designed is often difficult. Unneeded features might incur performance penalties. Or the component sometimes is just incompatible with the rest of the software.

The remedy to these problems is *horizontal scaling*. We want to create variations of a component so that it can meet many different needs. These variations can, for instance, mirror the design decisions during the creation of the component. Each decision can be seen as a *feature* of the final component, feature variations provide alternatives to the choices that have been made. Traditional implementation techniques face what Biggerstaff calls the *vertical/horizontal scaling dilemma* [Biggerstaff, 1997]: To increase code reuse, we want to scale a component vertically. To counter the negative effects of vertical scaling, we want to scale the component horizontally. But implementing every meaningful permutation of features and feature variations results in a combinatorial explosion of custom components. Singhal gives an example of this in [Singhal, 1996]: Booch's reusable library begins to exhibit signs of the

[1] object-oriented programming

Product	Lines of code
MS-DOS 1.0 (1981)	4,000
Microsoft Windows 3.1 (1992)	3 million
Microsoft Windows 95	15 million
Microsoft Windows 98	18 million
Microsoft Windows NT (1992)	4 million
Microsoft Windows 2000	35 million
Windows XP (2001)	45 million
Red Hat Linux 6.2 (2000)	17 million
Sun Solaris (1998-2000)	7-8 million

Table 1.1: The evolution of the Microsoft Disk Operating System exemplifies how software is growing in size (and therefore in complexity). The last two entries list the sizes of other operating systems, as a point of reference [Wheeler, 2001]

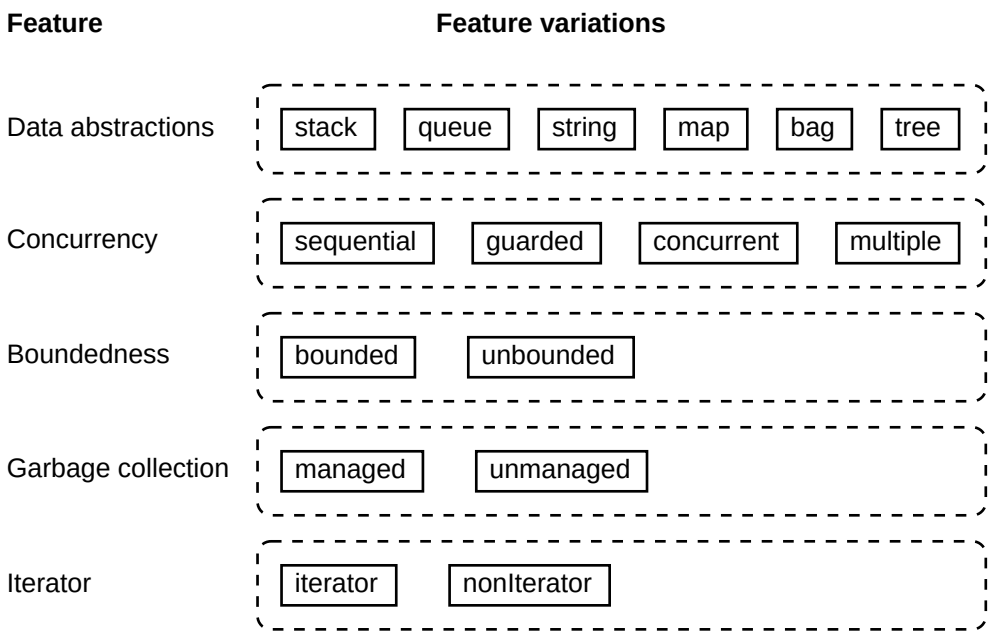


Figure 1.1: Horizontal or feature scaling of Booch Library

scaling dilemma even for comparatively small components. The library contains 17 abstractions that are mostly data structures such as stacks, queues and trees. These abstractions are organized according to four global features (figure 1.1):

1. Concurrency—if and how data inside a component is shared by multiple tasks. (4 variations)
2. Boundedness—can a component grow in size? (2 variations)
3. Garbage collection—how will data be garbage-collected? (3 variations)
4. Iterator—whether or not the data structure provides an iterator. (2 variations)

In addition to these global features, there are features that are specific to certain abstractions, such as deques or queues; for example:

- Balking—can an element be removed only from the front or back of a deque or queue or also from inside these data structures? (2 variations)
- Priority—is the deque or queue ordered by the value of a field of its entries? (2 variations)

Booch reports that there are 26 meaningful combinations of these features for queues. Introducing just one new feature (e. g., persistence) with two variations would mean that there are now 52 variations. Thus, because this library explicitly scales horizontally, it becomes quickly impossible to populate it using traditional programming paradigms.

Current research efforts concentrate on increasing the expressiveness of object-oriented languages to better handle horizontal scaling. These so-called *post-object programming* (POP) mechanisms provide the means to *decompose* software: We want to break it up into pieces representing features. Producing a custom-configured component is then a matter of *composing* the features it should have. Instead of having to populate the library with one *concrete* component (an instance of a composition) for every possible permutation of features, we just offer a set of features and let the client of the library decide what he needs. Features are also easier to understand than a monolithic body of software, because they are independent pieces of code dedicated to one purpose only.

In chapter 2, we present several POP programming paradigms and show what problems they solve. The first examples, *aspect-oriented programming* and *multi-dimensional separation of concerns*, specialize on decomposition. We argue that further domain-specific abstractions and optimizations that span several features are necessary to scale more successfully in both dimensions. These are areas *generative programming*, our last example, specializes in.

Chapter 3 introduces our solution, the *GenBorg generative programming environment*. GenBorg includes ideas from all of the paradigms of our survey and adds the following original contributions: Features are scaled further, non-code artifacts can be generated easily and there is a strong theoretical foundation for all these mechanisms, the *GenBorg algebra*.

Probe, a first tool of the GenBorg environment, helps in navigating and defining a structured base of software. One can also compose features or perform other transformations on the software and gets a visual representation of the result. The ideas behind Probe are explained in chapter 4. Chapter 5 contains a tutorial that demonstrates how the program is used in practice.

We summarize our contribution in chapter 6 and give an outlook on currently planned enhancements of GenBorg and Probe.

Chapter 2

Background

During the last few years, many ideas have been proposed to enhance object-oriented programming and design. In the following sections, we present three approaches and discuss what contributions they make: Aspect-oriented programming, multi-dimensional separation of concerns, and generative programming. We also give an example for an aspect-oriented programming mechanism, *mixin layers*, and introduce the generative programming system *GenVoca*.

In the next chapter, we present our solution to the issues raised by these post-object mechanisms, GenBorg, which is based on GenVoca.

2.1 Aspect-Oriented Programming

2.1.1 Problem

Aspect-oriented programming (AOP) recognizes that source code can be seen as an aggregation of *concerns*, parts of software that are relevant to a particular concept, goal or purpose [Ossher and Tarr, 2000]. Concerns are thus roughly equivalent to what we until now have called a *feature*. We would like to support horizontal scaling by encapsulating concerns in software units (which AOP calls *aspects*). Mixing and matching (*composing*) aspects could then produce feature-varied components.

Object-oriented languages are not flexible enough for encapsulating concerns which *cut across* their unit of encapsulation, the class. That means that concerns are often *scattered* and *tangled* in object-oriented software. The code of a concern is *scattered* if it does not reside in one place but is spread over the program. An example for this would be tracing functionality. It has to be added separately to each function one wants to observe. *Entanglement* occurs when several concerns overlap in one spot, e. g., when the same class of a drawing program takes care of both displaying and saving itself.

Tangling and scattering make it very difficult to retrofit functionality like tracing in programs or to eliminate it from them. We would also like to add or remove this kind of functionality *non-invasively*, without touching the source code, because the source might not be available and because it reduces the probability of introducing new bugs.

2.1.2 Solution

AOP languages eliminate tangled and scattered code by unifying all the statements related to a concern in one place and by *quantifying* them like this: *In program* P*, whenever condition* C *arises, perform action* A. Composing quantified statements with a program leads to the code of the action being (re)distributed throughout the program. Distribution is static and happens at compile-time, but the condition can contain dynamic elements such as a reference to the state of the call stack.

Quantification alone is not enough to reach the goals stated above, though: Traditional object-oriented

inheritance, for example, provides a limited form of quantification. If one edits a superclass, the changes get propagated to the subclasses. For this to work, the programmer has to be *cooperative*, he must follow an implicit contract (namely, to call every super method he overrides) that cannot be programatically enforced and is invasive. That is why AOP supports *obliviousness*, existing code does not have to be prepared for the addition of new functionality.

The language construct to implement quantification and obliviousness is called a *join point*. It is a certain well-defined point in the execution flow of a program and serves as a hook for adding functionality. The following elements provide the environment in which join points are used [Kiczales et al., 2001]:

- *Join point model*: Defines what join points are and how they can be described.
- *Pointcuts*: A means for selecting join points; the *condition* part of a quantified statement. It filters out a subset of all the join points in the program flow.
- *Advice*: The specification of behavior at a join point; the *action* part of a quantified statement.
- *Aspect*: The unit that encapsulates point cuts and behavior; the AOP version of packages or modules.
- *Weaving*: The process of attaching aspects to programs (and therefore of defining the *program* part of a quantified statement). This involves various decisions, especially if more than one advice is added to a join point: In what order are the changes to be applied? When? Should behavior be overridden? etc.

2.1.3 Examples

We have slightly modified a few examples from [Kiczales, 2001] of how the AOP concepts are implemented in practice. These examples are written in the AspectJ programming language which extends Java with AOP constructs.

```
pointcut intAccess():
    call(void Point.set*(int)) ||
    call(int Point.get*())
```

This pointcut named `intAccess` identifies all calls to methods of class `Point` that have an `int` parameter and a name that starts with `set` or that return an `int` and have a name that starts with `get`. The following advice references the pointcut we have just defined and adds behavior after each method call that the pointcut identifies.

```
after(): intAccess() {
    System.out.println("An int has been accessed!");
}
```

The final example defines an aspect that provides simple tracing by printing a message before certain operations occur in class `Display`.

```
aspect SimpleTracing {

    pointcut traced():
        call(void Display.update()) ||
        call(void Display.repaint(..));

    before(): traced() {
        System.out.println("Entering:" + thisJoinPoint);
    }
}
```

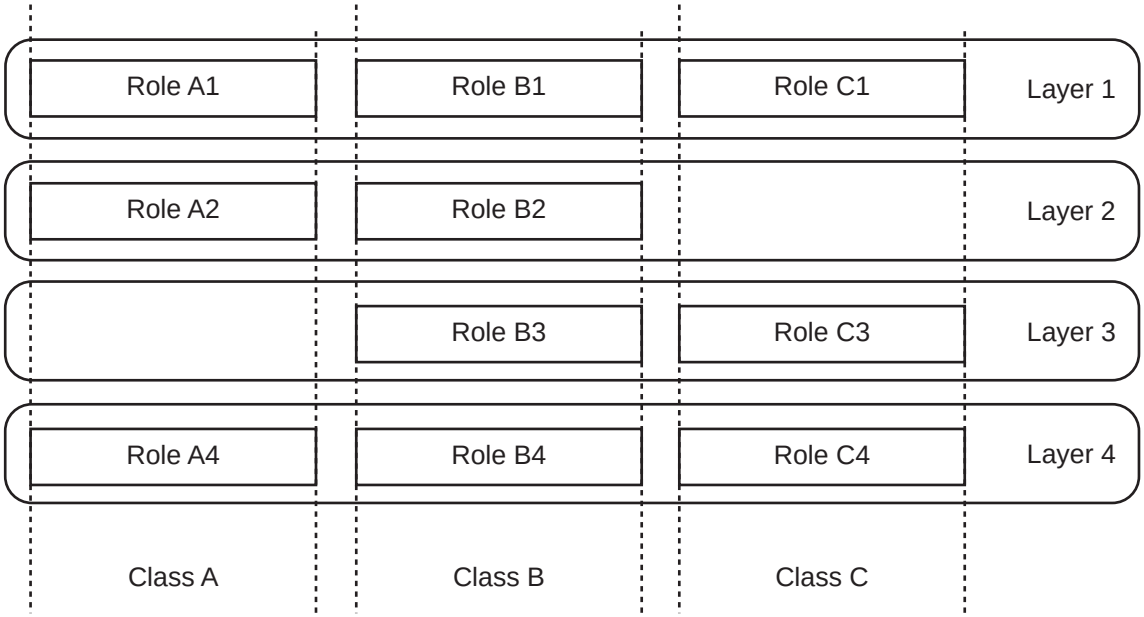


Figure 2.1: Example collaboration decomposition. Each of the classes A, B and C participates in several collaborations (layers). E. g., A plays role A1 in layer 1, role A2 in layer 2 and role A4 in layer 4

The aspect `SimpleTracing` encapsulates the pointcut `traced` and an advice. The pointcut selects all void methods in class `Display` that are either parameterless and called `update` or have any number of methods and the name `repaint`. The advice refers to pointcut `traced` and adds an output statement after everyone of the `traced` method calls. Other features of AspectJ are pointcuts filtering on properties of methods other than their signatures, join points not related to method calls, etc.

2.2 Mixin Layers

Although *mixin layers* [Smaragdakis and Batory, 1998] predate AOP, it is an AOP mechanism. We shall see later how mixin layer and AOP terminology correspond.

If one looks at OOP's smallest unit of abstraction, the object, one realizes that it is rarely self-sufficient. Instead, to implement a feature, several objects work together and adhere to a *protocol* (conventions about how to interact); they form a *collaboration*. The part of an object that implements a protocol is called the object's *role* in that collaboration. The same object can play a role in several collaborations (figure 2.1).

Thus, to unite in one place all the parts of the classes that participate in a collaboration, one has to *decompose* (break up) each class so that each piece plays exactly one role. Object-oriented programming languages already have a mechanism that can be used to take classes apart—inheritance. But inheritance poses two problems:

1. It breaks encapsulation and
2. you cannot arbitrarily mix and match the parts.

The reason for this is that (1) derived classes refer to the implementation of a superclass (2) with a static link. *Mixin classes*[1] [Flatt et al., 1998] solve the problem by generalizing inheritance. That is,

[1] also called *abstract subclasses*

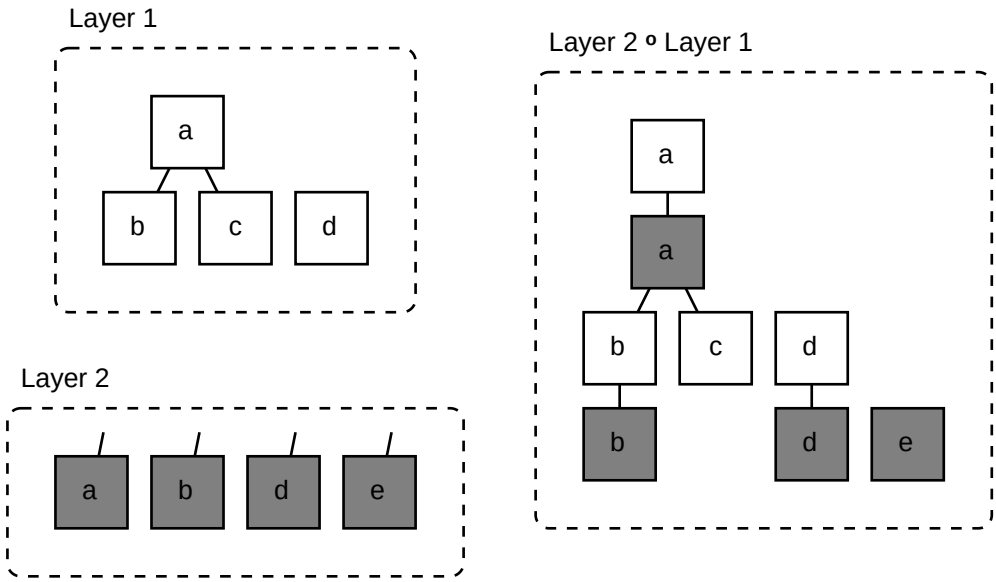


Figure 2.2: The new layer *Layer 2* ∘ *Layer 1* is created by composing *Layer 1* and *Layer 2*. Note how white b and gray a are both inserted after white a in the resulting inheritance chain. But gray a is part of a class called a and is therefore a direct successor of white a. Conversely, white b is part of class b, a subclass of class a, and inserted after the last addition to class a has been made.

a mixin is a template for a class whose superclass is specified via a parameter (1) that is typed by an interface (2). Eventually, one turns a set of mixins into a concrete class by filling in the superclass parameters.

A *mixin layer* is the construct that encapsulates a collaboration, each of the roles in the collaboration is implemented by a mixin class. To create a component, we *compose* (combine) the layers that implement the features we want that component to have, exactly like we do in aspect-oriented programming. Note that two composed layers are again a layer, only now the mixins that were originally part of the same class are joined (figure 2.2). Each class inside a layer has two coordinates: (1) What class it should be attached to when composing layers and (2) what role it plays in the collaboration. [Smaragdakis and Batory, 1998] simplifies that by standardizing both on the class name (classes refer to each other by name and each mixin class finally is part of a composed class of the same name). It should now be obvious how mixin layer constructs correspond to AOP mechanisms: A role is a join point, a mixin class is advice. A collaboration is a concern and its encapsulation, the mixin layer, is an aspect. The process of weaving corresponds to composing layers.

Just like a mixin class has the parameter *superclass*, a layer can be viewed as having a parameter *superlayer*. This is just one example of how mixin layer and mixin class are isomorphic which implies that we can build layers of layers etc. The next chapter will follow up on this idea of scaling layers.

2.3 Multi-Dimensional Separation of Concerns

Multi-dimensional separation of concerns (MDSOC) takes a different view on concerns than AOP: The definition of a concern is the same, but MDSOC recognizes that there are different kinds, or *dimensions*, of concerns. Examples for dimensions of concerns are:

- Data (classes in object-oriented programming)
- Features (printing, display)
- Aspects (persistence, logging etc.)

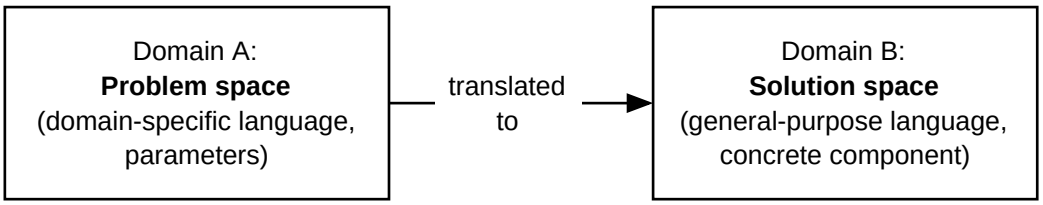


Figure 2.3: Generative programming translates between a problem and a solution space

- Configurations (specifying members of a family of programs, see the section on generative programming)
- Documentation
- Design rules (see next chapter)
- Performance properties

Most of today's programming languages are limited to decomposing along one dimension of concern and unable to express concerns that cut across dimensions, something Tarr, Ossher, Harrison and Sutton call the "tyranny of the dominant decomposition" [Tarr et al., 1999].

MDSOC defines a coordinate system that helps understand the notion of dimensions. First, we have to make some preparatory definitions: Software consists of *artifacts*, parts of software that are expressed in a (formal or informal) language. Each syntactic construct is called a *unit*. If the construct is atomic, it is called a *primitive unit*. Non-primitive units comprise other units[2] and are called *compound*. A *concern space* contains all units of a body of software plus means for identifying, encapsulating and integrating (composing) concerns. These mechanisms thus aid in organizing units by separating concerns.

Hyper/J, an augmentation of Java that supports MDSOC [Ossher and Tarr, 2000], uses a concern space that is called *hyperspace*. It is a multi-dimensional matrix. Each axis represents a dimension of concerns, each concern is a point along the axis. As an example, classes, the dominant decomposition of object-oriented programming are points along the "data" dimension of concerns. Units are placed into this coordinate system according to what concerns they affect. One coordinate per dimension names the affected concern (or "none" if this dimension is unaffected). Therefore, each dimension partitions the units according to a decomposition scheme and each concern is a hyperplane containing every unit relevant to it. Hyper/J encapsulates arbitrary hyperplanes as *hyperslices* which are analogous to aspects. Hyperspaces are convenient in that they explicitly state the dimensions of interest, what concerns are part of a dimension and that each unit belongs to exactly one concern per dimension (or none).

Two properties distinguish Hyper/J from AOP languages as exemplified by AspectJ: First, AspectJ has one base class hierarchy, a *model*, that is refined by aspects. Aspects typically can only be understood in their relation to the model. It is also impossible to integrate two aspects to form a new one. In contrast, hyperslices are independent, arbitrarily composable, and not restricted by a common base hierarchy. Second, Hyper/J permits *on-demand remodularization*. One can simultaneously decompose along different dimensions, which is like having several overlapping (read *and* write) views on the software.

2.4 Generative Programming

Giving a concise definition of *generative programming* (GP) is difficult, because the term is used very broadly throughout the literature. Generative programming is a compilation process between

[2] primitive and compound

two domains: A *problem space* is translated to a *solution space* (figure 2.3). Domain information is specified in a language specific to that domain. Further examples below will clarify this notion, but first we want to quote the definition of generative programming given by [Czarnecki et al., 1998].

> "*Generative programming* is about modeling families of software systems by software entities such that, given a particular requirements specification, a highly customized and optimized instance of that family can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge."

The goal of generative programming is to increase code reuse and evolvability of software by providing the means for designing *domain-specific abstractions* (the ability to state the problem in terms of the problem domain). Two difficulties involved in this are:

1. the semantic gap between domain-specific abstractions and general-purpose programming languages and
2. run-time perfomance penalties incurred by giving a system flexibility and generality.

Generative programming eliminates these difficulties by separating the problem space (parameters) from the solution space (output) and by translating between them: The parameters of the generation process are domain-specific, solving (1). The output is usually static customized code that can be performance-optimized during the translation process. We have thus traded generation-time performance (and flexibility) for run-time performance and solved (2). Let's look at two generation processes and how generative programming borrows from other programming paradigms to ease implementing them.

Domain-specific languages (DSL) use the vocabulary of a problem domain to specify a solution, therefore bridging the semantic gap mentioned above. Examples are markup languages and languages for algebraic manipulation (like Mathematica). GP translates between the DSL (the problem space) and an executable program (the solution space), usually by either interpreting the DSL (like HTML in web browsers) or by compiling it (like Java server pages). Representing knowledge this way has the advantage that domain-related optimizations are possible that cannot be achieved otherwise. Example: In matrix algebra, $A \cdot E$ (multiplying a matrix A with the identity matrix E) can be simplified to just A. This optimization is not possible (or rather, not recognizable) once this operation has been compiled to, say, C code. Note that optimization is also a translation and that both problem and solution space are the DSL. The generative programming system *Draco* [Neighbors, 1989] supports optimizing translations from a domain to itself.

Generative programming is also used to create customized components. A user parameterizes a component, e. g., by specifying what features it should have, and a GP system generates it. In the case of parameters specifying features, aspect-oriented programming can be used by GP to integrate the corresponding aspects. Going beyond AOP, GP is able to make optimizations that take properties into consideration that are global to all parts making up the output. AOP can only optimize locally, inside an aspect. As an example, think of a component implementing a dictionary data structure. Given a parameter specifying whether the entries should be ordered by their key or not, generative programming can *automatically* choose how they are stored: As a hashtable that is faster for most applications or as a tree that is necessary for ordering the keys. AOP would help GP by giving it two alternative aspects for data storage. Because data storage is specified abstractly ("order the keys!") and not concretely ("use a binary tree!"), a GP system can transparently substitute other, more efficient, implementations for the tree aspect, once they become available. Generating components neatly solves the problem of combinatorial explosion of custom components mentioned in the introduction, without compromising efficiency: 3 features, with 3 variations each, can be implemented as 9 aspects instead of 27 custom components (or some dynamic, but inefficient, mechanism). [Batory et al., 1993] have presented a GP system that produces data structures in this manner.

We would like to introduce two more terms: *Families of software* are components or applications that are built from a common set of assets [Weiss and Lai, 1999]. A *Product line* consists of the assets used to produce the family. Therefore, generative programming can also be regarded as a tool for

implementing product lines. None of the translation processes we have seen are mutually exclusive and are often used together: [Batory et al., 2000b] give an example of components built using a domain specific language for state machines.

2.5 GenVoca

GenVoca is an example of a generative programming system. It is a design methodology (and a set of tools to support this methodology) for creating product-lines and building architecturally extensible software—i. e., software that is extensible via component additions and removals.

GenVoca is based on scaling the well-known programming methodology called *stepwise refinement* that originated in the writings of Wirth and Dijkstra in the late 1960s/early 70s [Wirth, 1971]. It advocates writing of efficient programs by progressively revealing implementation details. Traditional notions of refinement are both very concrete and only being used at a small scale; an example refinement is to replace a literal with a subroutine call. Programming this way, even small programs necessitate numerous refinements. GenVoca abstracts stepwise refinement and scales it to a component or layer granularity, so that each refinement adds a feature to a program, and composing a few refinements yields an entire application.

The key is to regard programs as constants and refinements as functions that add functionality. Continuing the example for a traditional refinement, let a represent a program with a literal. The refinement replacing the literal with a subroutine call (and defining the subroutine) is a function $f(x)$ that takes a program as input and produces a refined program as output. $f(a)$ represents the result of refining a. But refinements need not be restricted to simple substitution, each refinement could add a feature or even several features. Consider the following constants that represent programs with different features:

```
b         // program with feature b
c         // program with feature c
```

Each of the following refinements adds a feature:

```
g(x)      // adds feature g to program x
h(x)      // adds feature h to program x
i(x)      // adds feature i to program x
```

A multi-featured application is then an *equation* that assigns a name to a composition of functions. Different equations define a family of applications, such as:

```
app1 = g(b);          // app1 has features b and g
app2 = h(c);          // app2 has features c and h
app3 = g(h(i(b)));    // app3 has features b, i, h and g
```

The set of all functions and constants that are available for program construction is called a *model*. A model and the set of equations that it can produce constitute a *product line*. Note that in this paradigm, each function not only defines a feature but also its implementation. This enables us to represent distinct implementations of the same feature by different (e. g., indexed) functions:

```
j1(x)         // adds feature j (with implementation 1) to x
j2(x)         // adds feature j (with implementation 2) to x
```

To optimize the performance of a program that needs feature j, one therefore needs to rewrite its equation so that it contains the optimal j_i for the task at hand. [Batory et al., 2000a] shows that it is thus possible to automatically create software (or rather, its equation) that is optimal with regard to some qualitative criteria, given a set of declarative constraints for a target application.

As a refinement typically cannot be applied to arbitrary programs and not all combinations of features make sense, GenVoca also provides the means to constrain function applications both semantically and syntactically [Batory and Geraci, 1997]. A typical syntactic constraint is that a program must implement a set of well-defined Java interfaces, semantic constraints can require the implementation of the interfaces to satisfy certain properties.

One of the advantages of the GenVoca paradigm is its generality. Refinements can be implemented in a number of ways, e. g., as mixin layers or in a domain-specific language (Batory et al. give an example of a domain-specific language for state machines in [Batory et al., 2000b], but the DSL might even describe something non-executable like documentation). In the next chapter we will present GenBorg, another generative programming system that extends GenVoca's ideas in several interesting ways.

Chapter 3

GenBorg

We present our answer to the difficulties of scaling software: The generative software environment GenBorg. We introduce it informally and then define a theoretical foundation in the form of an algebra (section 3.2).

3.1 Informal Introduction

When working with GenVoca, it became apparent that there were two areas where one would like to improve it, while retaining its simple theoretical and practical elegance:

1. *Scale:* The common way of implementing a GenVoca refinement, a mixin layer, operates on a scale that is too small for some applications. On the other hand, when composing mixin layers, we are not only refining at layer, but also at sub-layer level where classes extend each other. Both this increase and decrease in scale when applying a refinement should be practically supported and theoretically described in a generative software environment.
2. *Support for non-code artifacts:* Most efforts regarding separation of concerns have so far concentrated on code. But what about other, non-code, artifacts such as documentation, design rules, performance properties, or language-related resources? They are also part of a body of software and therefore should not be excluded in a programming paradigm. We seek a model that allows the automatic generation of code as well as of non-code artifacts.

The following sections elaborate these points.

3.1.1 Scaling Refinements

A GenVoca model allows one to specify and generate a subsystem or an application, but cannot generate *several* programs that cooperate with a common goal. These *application suites* are becoming increasingly important in software engineering. A classic example of an application suite is an "Office" product: A word processor, a spreadsheet program and a presentation program work together to create documents that contain a mix of textual, tabular and graphical data. Concerns such as support for file formats cut across applications and are called *facets*.

To further illustrate the idea of a facet, we look at a development environment for customized versions of the Java programming language that consists of a compiler, a document generation tool and a debugger. The base version of this application suite only implements pure Java and is therefore equivalent to the *Java Development Kit* tools `javac`, `javadoc` and `jdb`. If we extend Java to support matrix arithmetic, we have to simultaneously refine every member of the suite: the compiler, documentation generator and debugger need to handle new language constructs for manipulating matrices. This kind of simultaneous refinement is an example of a facet. If each program is constructed from

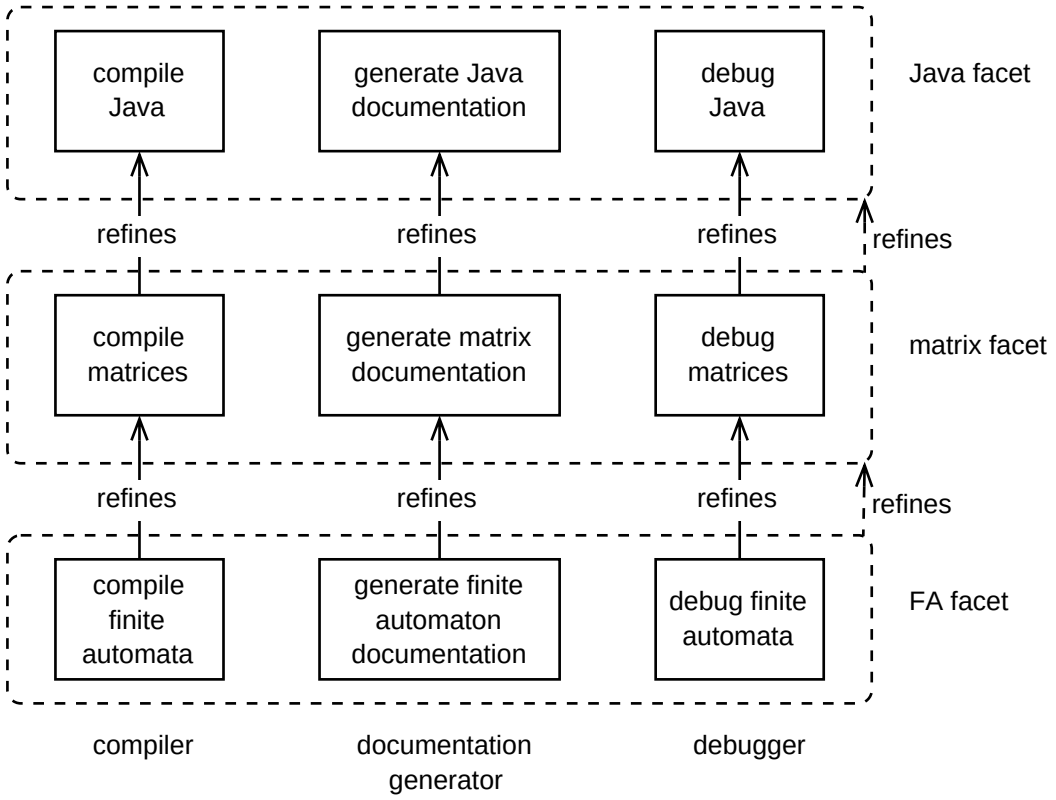


Figure 3.1: This development kit is an example of an application suite. It has been created by composing three facets and understands an extended version of Java with native support for matrices and finite automata (FA).

a set of layers, a facet just adds a layer to it and can thus be represented as a set of layers. The original programs are the base facet `Java`, the layers to support matrices are contained in the `matrix` facet. Extending Java so that it natively understands finite automata results in similar refinements, which we package as the `finite automata` facet. We can now compose these three facets to yield a *family* of four different application suites: The development kit supports either pure Java, Java with matrices, Java with finite automata, or Java with matrices and finite automata (figure 3.1).

The next step is to find out how we can express a facet theoretically. GenVoca's accomplishment was to represent refinements as functions and applications as equations. The idea of a refinement cleanly scales to application suites. A model for a family of collaborating programs is a set of facets. Each facet is either a base suite represented as a constant or a refinement of a suite represented as a function. An equation describes an instance of the model, i. e., a concrete application suite.

While we can represent a facet as a function, it is also a set of functions, namely, refinements implemented as layers. One kind of layer, a mixin layer, has a structure that is similar to a facet (compare figure 2.1 and 3.2): Both mixin layers and their constituents, mixin classes, are representable as functions. This pattern of a refinement's double duty as a function and a set of functions continues if we take mixin classes to be sets of methods[1] and method overriding to be a form of refinement. Therefore, facet, layer and class form a hierarchy with facet at the root, where each level is similar in structure to every other level (figure 3.3), a property that we call *self-similarity*[2]. GenBorg provides the tools to adequately describe this kind of self-similarity. Following the discussion of the introduction, being able to use facets in GenBorg should improve code reuse, because the size of a refinement has increased compared to GenVoca and aspect-oriented programmming (where a refinement is implemented as an aspect).

[1] A data member can be regarded as a combination of two methods: One method for reading the value of the data

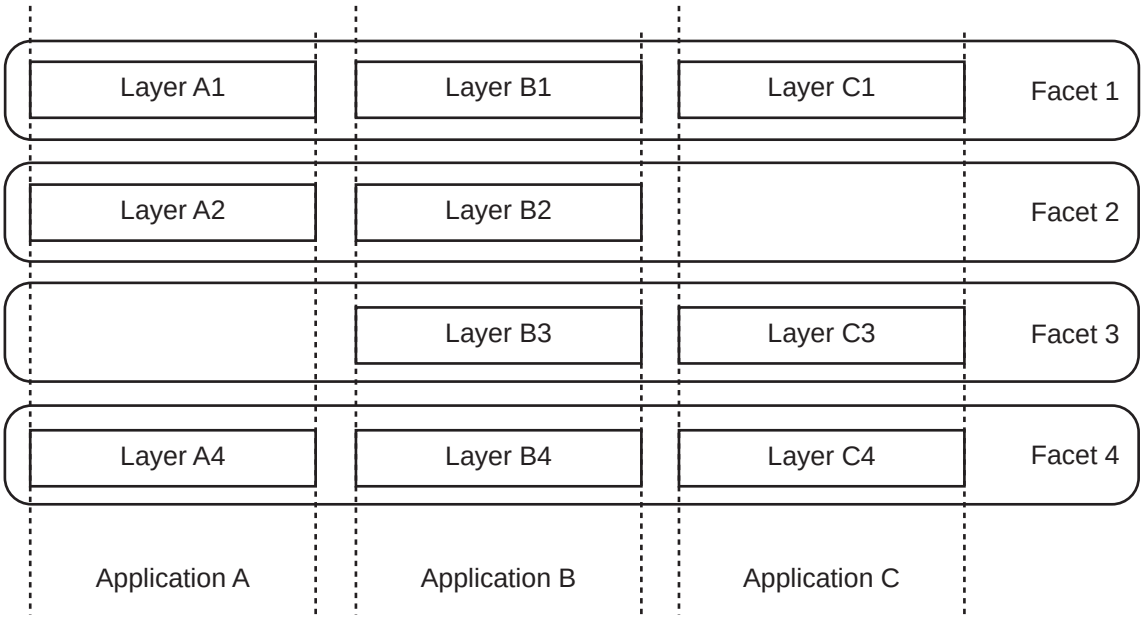
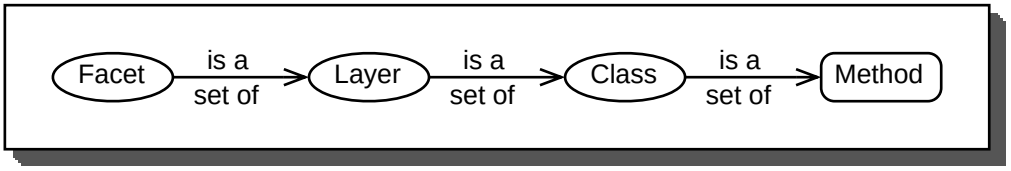


Figure 3.2: Decomposing applications by facets.



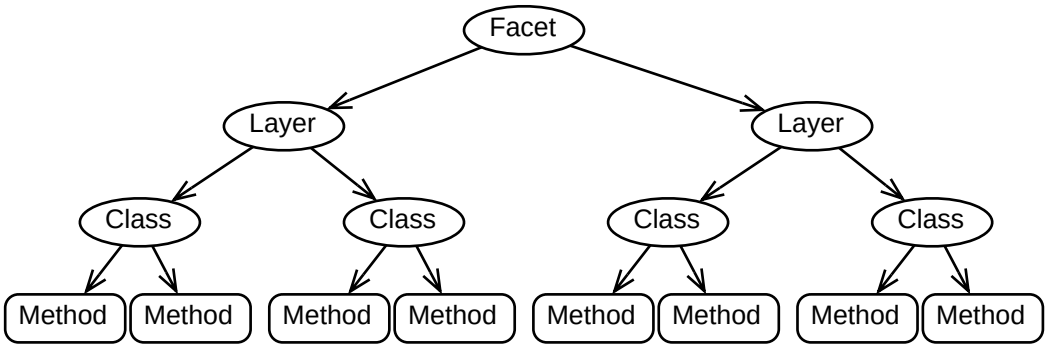


Figure 3.3: A facet is a self-similar hierarchy: At every level, the same structure is repeated—facet, layer and class are at the same time a function and a set of functions.

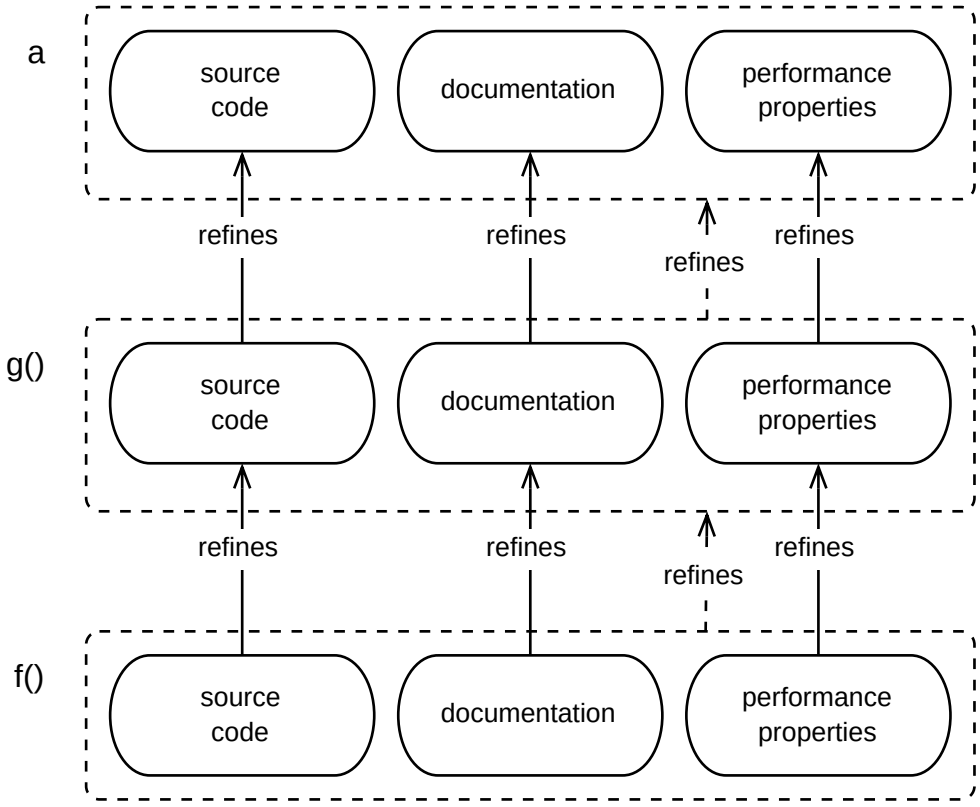


Figure 3.4: Composing the related artifacts code, documentation and performance properties for the equation $f(g(a))$ in parallel, instead of generating each one separately.

3.1.2 Generating Non-Code Artifacts

There is a dimension of concerns called "refinement". Until now, we have only looked at one member of the hyperplane at each refinement, source code. But many non-code artifacts such as documentation, formal properties and performance properties are related to the code artifact implementing the refinement. A difficulty arises when composing only the code artifacts: One also has to update the related non-code artifacts in parallel. If, for example, the GenVoca constant a and the refinements $f()$ and $g()$ have the components code (a_{code}, $f_{code}()$, $g_{code}()$), documentation (a_{doc}, $f_{doc}()$, $g_{doc}()$) and performance properties (a_{perf}, $f_{perf}()$, $g_{perf}()$), then when generating application code $app_{code} = f_{code}(g_{code}(a_{code}))$, one has to generate the documentation $app_{doc} = f_{doc}(g_{doc}(a_{doc}))$ and the performance properties $app_{perf} = f_{perf}(g_{perf}(a_{perf}))$, too. The problem of course is, that these other artifacts must be kept in sync manually (which itself is costly and error-prone). What we would like our generative software environment to do, is to evolve related artifacts automatically and in parallel (figure 3.4).

3.1.3 Collectives

From now on, we will stop differentiating between functions and constants. In our algebra, there will be only one kind of entity, called *unit*. It is the algebraic representation of a facet, a layer, a class, a documentation file etc. Instead of applying functions such as $f(g(a))$, we *compose* units using a composition operator $\circ$ as in $f \circ g \circ a$. This is mathematically equivalent and better expresses the kind of symmetry we have found between units that represent the same kind of artifact. It also allows us to introduce other operators in the future. Applications that before were naturally impossible, because

member and one for changing it.

[2]Other examples for self-similar constructs are wavelets, probability constructs and fractals.

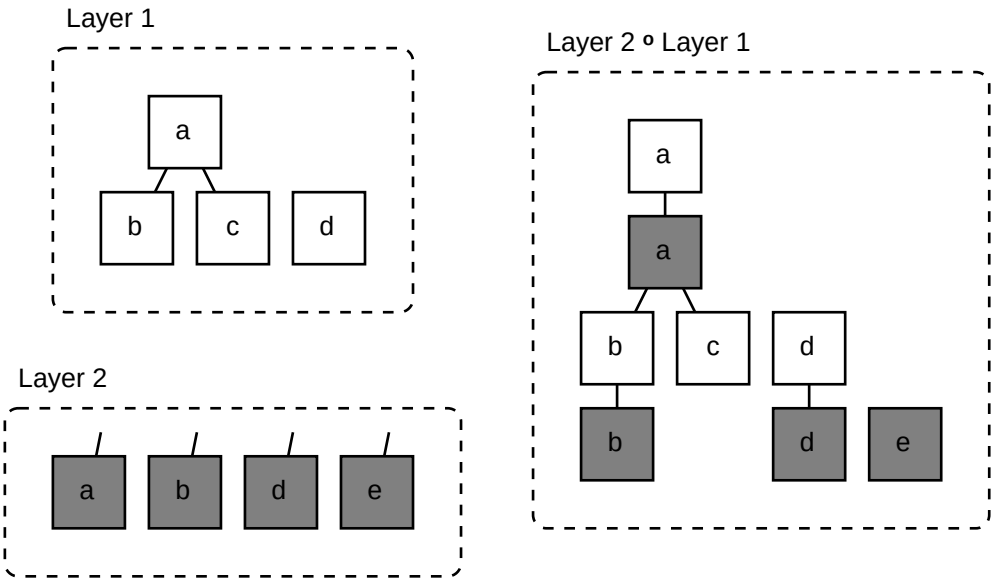


Figure 3.5: Composition of layers, as introduced in figure 2.2 on page 10.

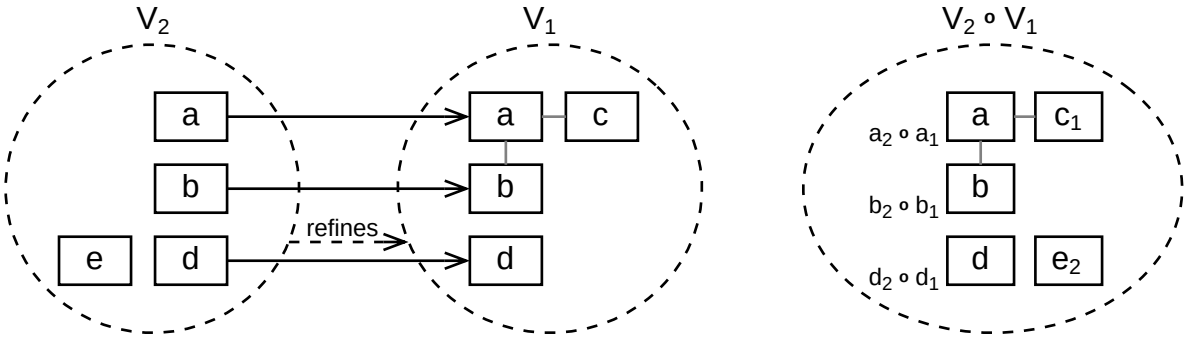


Figure 3.6: The composition of layers from figure 3.5 represented as a composition of the collectives V_2 and V_1. Each rectangle displays the role of a unit, the indices are only shown to indicate the source of units. Where necessary, we annotated units with their equation. The gray line between a_2 and b_2 denotes the inheritance relation which is not expressed inside the algebra.

constants could not be applied to other entities, can be prevented through GenVoca's design rules which fill the important need to describe semantic constraints in GenBorg.

The algebraic construct that allows us to both scale layers (up and down) and to evolve different dimensions of a refinement in parallel is called a *collective*. A collective is a set of units (which are sometimes called the *subunits* of the collective). What makes a collective so powerful is that it is a unit, too, mirroring the double (self-similar) nature of facets, layers and classes we have discovered above. Therefore, collectives can be nested and every collective is a tree whose inner nodes are collectives and whose leaves are *primitive* (non-collective) units. How does this solve our problems? Let's look at each one in turn.

As the abstraction of, for instance, a layer is a set of functions (units), it is obvious that one can model its structure using nested collectives. What is missing is the means to compose collectives in a way that preserves the semantics of composing layers. First, we have to define how primitive units are composed. This depends on what kind of artifact is represented by the unit. For example, if units m_2 and m_1 correspond to methods, composing them results in m_2 overriding m_1. Composition of the classes c_2 and c_1 leads to c_1 becoming the superclass of c_2. Second, whether we compose two facets,

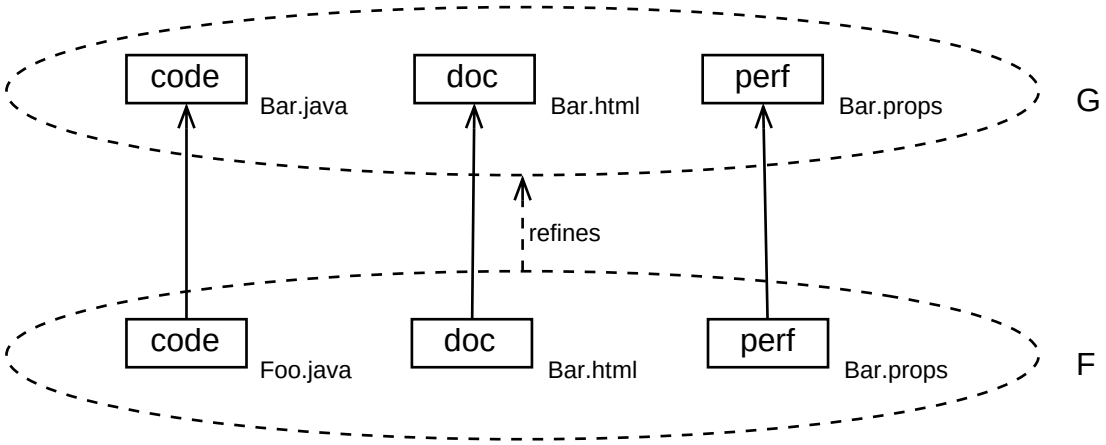


Figure 3.7: Composing the collectives, $F \circ G$, refines the dimensions *code*, *documentation* and *performance properties* automatically and in parallel. Each rectangle denotes a unit and is tagged with an index indicating the file represented by the unit.

two layers or two classes, we always have to make sure that the members of the refining entity are applied to the correct members of the refined entity. So, to be able to compose two collectives V_2 and V_1, we need to know what units of V_2 and V_1 should be composed in the result set $V_2 \circ V_1$. To that effect, we use the same mechanism as in mixin layers and tag each element in a collective with a *role*, the GenBorg version of a join point. Roles direct what units of V_2 refine units of V_1. $V_2 \circ V_1$ contains a composition $v_2 \circ v_1$ for every pair of units $v_2 \in V_2$ and $v_1 \in V_1$ that have the same role. The remaining subunits are added as they are, uncomposed. We have to recurse this algorithm for non-primitive units v_2 and v_1. Representing the composition of layers in figure 3.5 as a composition of collectives, we get the exact same result as before (figure 3.6). If collectives represent facets, roles indicate what application a subunit (which is a layer) belongs to; in classes, method type signatures (as, e. g., defined in the Java language report [Gosling et al., 2000]) are used as roles. We have thus reached our first goal and provided a scalable, generic mechanism for composing sets of units that subsumes the constructs used in GenVoca. Note that we currently constrain units to play exactly one role and roles to be played by at most one unit per collective. Dropping this constraint is the subject of ongoing research.

Similar to a mixin class in a layer, a unit has actually two coordinates of identification: A "horizontal" identifier among the fellow members of the collective (the role of a mixin in a collaboration) and a "vertical" identifier that brings the refinement and the refined together (the name of the class a mixin eventually becomes part of). Part of the elegance of GenBorg arises from the fact that the role serves both purposes, i. e., we normalize the two coordinates to one.

Returning to the problem of co-evolving multiple dimensions of the same refinement, we find that we can solve it by again using collectives and roles. Take two layers consisting of a set of classes (Java source code as primitive units). In order to support more dimensions than just source code, we replace these primitive units by collectives that contain documentation and performance properties in addition to the code. When a class collective F refines a class collective G, we want the code of F to refine the code of G, the documentation of F to refine the documentation of G etc. (figure 3.4). This can be achieved by giving code the same role in G and F and by doing the same for documentation and performance properties (figure 3.7). In a composition of class collectives, every dimension of a refinement is composed automatically and in parallel—which is what we sought to achieve.

3.2 GenBorg Algebra

In the following sections, we define a formal algebraic foundation of the concepts we have informally introduced in the last chapter, namely, the GenBorg algebra which consists of a data structure called

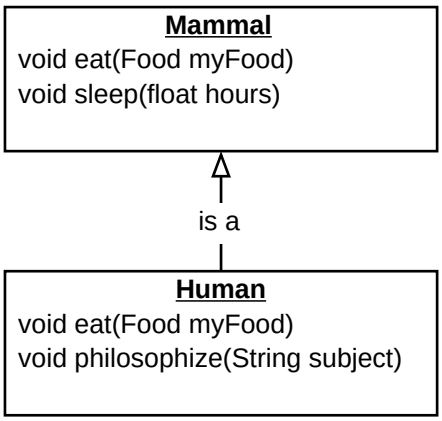


Figure 3.8: Inheritance between a superclass `Mammal` and a subclass `Human`.

Java	GenBorg
all methods and classes of a program	unit space
inheritance	composition
class	collective
subclass relation	subtype relation
method	unit
type	type
signature	role

Table 3.1: Corresponding concepts in Java inheritance and GenBorg composition

unit and one operator, *composition*. We tried to stay loosely compatible with the terminology used in section 2.3.

3.2.1 Review

As an introduction to the concepts we are about to define, we review something that the reader should already be familiar with and present it in the terms of the GenBorg algebra: Inheritance of classes in Java. Take the example of superclass `Mammal` being extended by subclass `Human` (figure 3.8). We can also view subclassing as composition where both the superclass and the subclass are a set of methods (figure 3.9). Note that the name `Human` stands for both the refinement and the result of the composition. When combining the sets of the two classes, methods in `Human` *override* (refine) methods in `Mammal` that have the same *signature* (method name plus the types of the arguments). A requirement that is always trivially fulfilled by inheritance is that a refinement has to be a subclass of its argument. GenBorg demands that this condition hold for collectives that are to be composed. Table 3.1 compares the terminology of Java inheritance and GenBorg composition.

3.2.2 Units

Any body of software can be decomposed into atomic pieces (what exactly constitutes atomicity depends on the language they are expressed in and on the application of the algebra). We call these pieces *primitive units*. A *compound unit* is a set of primitive and compound units. Compound units are also called *collectives*. As a collective can contain any kind of unit and is itself a unit, progressive nesting of sets of units results in a tree of units whose inner nodes are collectives and whose leaves are primitive units. The root of such a tree is called the *root unit*. The units contained in a collective V are *subunits* or *child units* of V; *direct* subunits are only members of V (and not of a collective that is nested in V). Additionally, each unit in a collective has a *role* that is unique in that collective. These

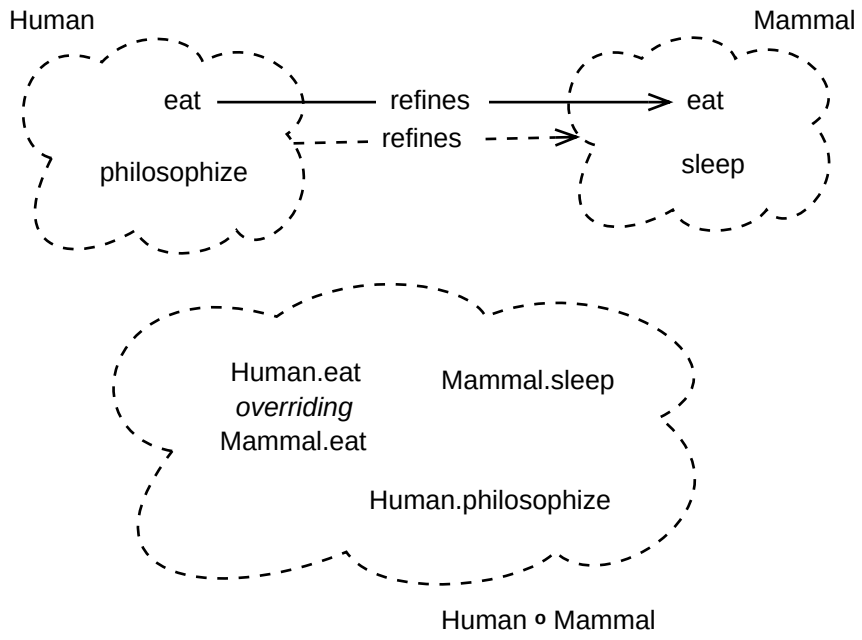


Figure 3.9: The inheritance of figure 3.8 represented as composition

concepts are expressed in the following definition.

Definition 3.1 (Unit space). A unit space $\mathcal{U}$ is a tuple $(U, R, u_{root}, \text{role})$ where

1. U is the set of all units in a body of software. A unit is either *primitive* or a set of units $V \subset U$, in which case V is called *compound* or a *collective*. A collective cannot be an element of itself[3].
2. R is a set of *role symbols*.
3. $u_{root} \in U$ is the *root unit*, a specially designated unit that contains all other units.
4. $\text{role} : U \to R$ is a function that assigns each unit a role. There cannot be two units with the same role in one collective.
5. $\text{roles} : C \to 2^R$ returns the set of roles of the subunits of a collective V. It is defined as $\text{roles}(V) := \{\text{role}(v) | v \in V\}$.
6. $C_{\mathcal{U}} \subset U$ is the set of all collectives in U. We usually omit the index and just write C if the context is clear.

3.2.3 Types

We now introduce a type system for units that is motivated by type systems of object-oriented languages. We assign each unit a type, which is usually related to what language the unit is expressed in and what is being described. It is used for expressing whether units can be composed or not. Types are partially ordered through a subtype relation.

Definition 3.2 (Typed unit space). A typed unit space $\mathcal{U}$ is a tuple $(U, R, u_{root}, \text{role}, T, \text{type}, \text{subof})$ where $U, R, u_{root}, \text{role}$ are defined as above and

1. T is a set of of *type symbols*.
2. $\text{type} : U \to T$ is a function that assigns each unit a type.

[3]This guarantees that a collective has a tree structure.

3. subof $\subset T \times T$ is the *subtype relation*, a reflexive partial ordering which can be a forest. No primitive unit can have a type that is a subtype of the type of collective and vice versa (i. e., primitive and compound types are disjoint).

3.2.4 Composition

A *model* contains a unit space and introduces composition for primitive units. We will see shortly how composition of primitive units induces a composition for all units.

Definition 3.3 (Model). A model $\mathcal{M}$ is a tuple $(\mathcal{U}, \circ)$ where

1. $\mathcal{U} = (U, R, u_{root}, \text{role}, T, \text{type}, \text{subof})$ is a typed unit space.
2. $\circ : (U - C) \times (U - C) \to (U - C)$ is the *composition* operator for primitive units ($U - C$ is the set of primitive units; we use set-theoretical difference to express the fact that every unit that is not a collective is a primitive unit).
3. The type of a composition has to be a subtype of the right operand's type: $\text{type}(v \circ w)$ subof $\text{type}(w)$, where $v, w \in U$ (we use the more general set U instead of $U - C$ as a preparation for extending composition of primitive units, below). The reason for this constraint is that we want a refined unit to stay compatible with the original unit.

Composition of primitive units can apply to any kind of unit. The idea is that once one has defined composition for primitive units (say, composition for similar artifacts—how to compose code, how to compose documentation, etc.), composition of collectives follows naturally. This kind of canonical composition of collectives is called *composition by role*.

Definition 3.4 (Composition by role). Given a unit space $\mathcal{U} = (U, R, u_{root}, \text{role}, T, \text{type}, \text{subof})$, composition by role $\circ$ extends composition of primitive units as follows ($v, w \in U$): The result $V \circ W$ of composing them is the union of the following sets.

1. Units whose role exists in both collectives appear as a composition in the result:

$$\{v \circ w \mid v \in V, w \in W \wedge \text{role}(v) = \text{role}(w)\}$$

2. Units in V whose role is not element of $\text{roles}(W)$ are simply copied:

$$\{v \mid v \in V \wedge \text{role}(v) \notin \text{roles}(W)\}$$

3. The same applies to the units of W:

$$\{w \mid w \in W \wedge \text{role}(w) \notin \text{roles}(V)\}$$

3.2.5 Conclusion

In this chapter, we've seen how the software environment GenBorg elegantly solves many difficulties related to software reuse. A theoretical foundation, the GenBorg algebra, clarifies the notions of units and composition by giving precise mathematical definitions. The next two chapters will demonstrate how GenBorg's concepts can be applied practically. They introduce *Probe* , an implementation of GenBorg, by first explaining the ideas behind it and then walking through a tutorial.

Chapter 4

Probe

Probe is a program that uses a graphical user interface to perform and visualize compositions of a GenBorg model and its instances. In this chapter, we introduce the main concepts behind Probe. The next chapter is a tutorial that shows how to use Probe by walking through a concrete example.

Note that currently, Probe's primitive units are artifacts (i. e., files). We've imposed this constraint to simplify the initial design, but GenBorg can handle cases more general than that, for example method units that are a fraction of a `.java` file. The type system is also relatively simple and mainly provides a syntax-based mechanism for enforcing the difference between functions and constants, a legacy of a previous version of the algebra.

4.1 Defining a Model

Ideally, we want models to be as simple to define as possible. It turns out that the most natural way of expressing the tree-structured unit space of a model is to encode primitive units as files and collectives as directories. Consider a model whose root unit contains a set of layers that in turn consist of mixins comprising at most one code and one documentation artifact (figure 4.1). To define it using the file system, we create one directory per mixin that contains code and documentation for this mixin. Several mixin directories are grouped by the directory of a layer, the model directory contains zero or more layer directories.

A model directory's nested subdirectories are almost everything that is needed to define a unit space. The rest of a model's data is specified in a *schema*. It also contains information that helps in translating between[1] directories and a unit space. The schema is encoded in the *Extensible Markup Language* (XML) and stored as a file named `schema.xml` in the model directory. XML's tree structure makes it a good choice for expressing information about unit spaces. The main purpose of the schema is to identify *categories* of similar units and to specify their properties, such as their types. `Layer`, for example, is a category in the above model. A category describes what units it comprises by specifying the files they are based on. We'll walk through a series of examples to introduce the elementary features of schemas. The most basic schema file for our example model describes the file structure of the model directory (figure 4.2) as a tree of categories.

```
1  <schema>
2      <directory category="Model">
3          <directory category="Layer">
4              <directory category="Mixin">
5                  <artifact category="Code">
6                  </artifact>
7                  <artifact category="Doc">
8                  </artifact>
```

[1] back *and* forth, as we shall see later.

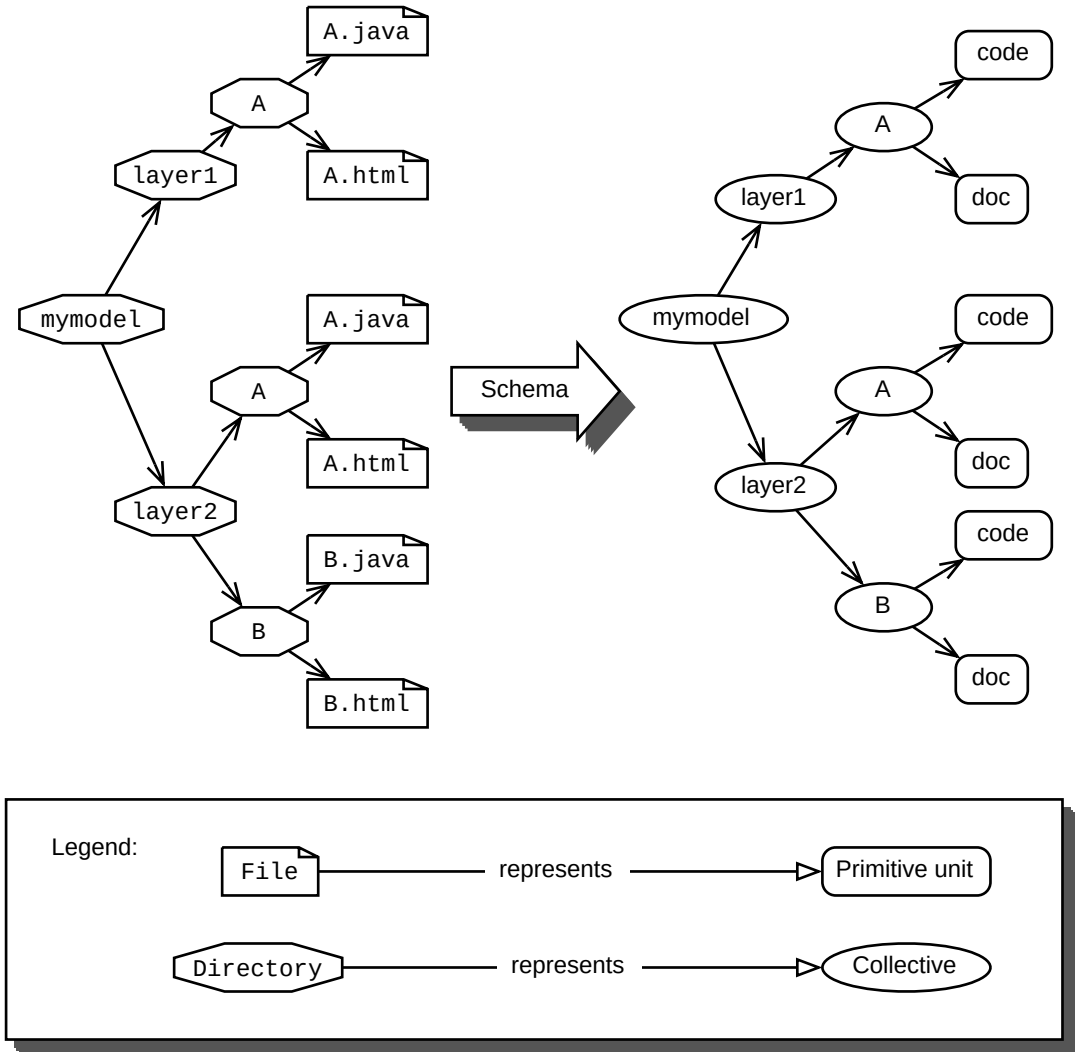


Figure 4.1: There is a straightforward mapping between the contents of a model and its storage format, the *model directory*: Directories store collectives and files store primitive units.

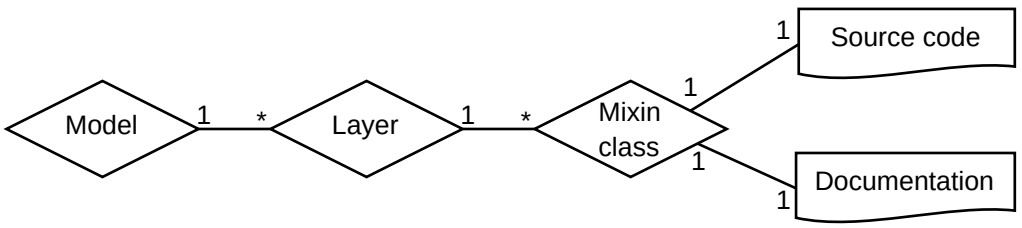


Figure 4.2: The structure of the example model directory expressed as a tree of file *categories*: A model directory contains a set of layers. Each layer directory contains one or more mixin classes with a code and/or documentation file each.

```
 9              </directory>
10          </directory>
11      </directory>
12  </schema>
```

We see that each node of the schema tree defines a category of units by stating what files these units are based on: Units in category `Model` are based on directories (line 2), units in category `Code` are based on artifacts (line 5) etc. To further restrict what files belong to a category, one can use patterns that are *matched*[2] against file names. The mapping of files to categories is actually bijective; we'll see in section 4.5 why this is necessary. Our schema with file patterns looks like this:

```
<schema>
    <directory category="Model" file="*">
        <directory category="Layer" file="*">
            <directory category="Mixin" file="*">
                <artifact category="Code" file="*.java">
                </artifact>
                <artifact category="Doc" file="*.html">
                </artifact>
            </directory>
        </directory>
    </directory>
</schema>
```

The file patterns are specified through the XML attribute `file`. The star in strings such as `"*"` is used as a wildcard, just like in Unix or MS-DOS shell programming. Used on its own, it matches any string, prepended by a literal string p it matches any string whose prefix is p, appended by a literal string s, it matches any string whose suffix is s. Therefore, *any* subdirectories of a layer directory belong to category `Mixin` and `.java` files in a mixin are all part of category `Code`. Category `Code` and `Doc` pose a problem, though. There should be only one code and one documentation file in a mixin. The base of both file names must be the same as the role of the mixin (so that code in mixin `A` is stored in the file `A.java`). We'll be able to enforce this restriction later. By adding the XML attribute `role` to a category declaration, we can specify role names for the units in this category:

```
<schema>
    <directory category="Model" file="*" role="*">
        <directory category="Layer" file="*" role="*">
            <directory category="Mixin" file="*" role="*">
                <artifact category="Code" file="*.java" role="code">
                </artifact>
                <artifact category="Doc" file="*.html" role="doc">
                </artifact>
            </directory>
        </directory>
    </directory>
</schema>
```

Because there will be only one code and one doc artifact per mixin, they play the constant roles *code* and *doc* (similar to figure 3.7 on page 20). The role specification of category `Layer`, on the other hand, is variable and refers to whatever string has been matched by the wildcard in the `file` attribute. This means that the role name of a layer unit is the same as the name of the directory on which it is based. The above schema turns the following file structure (taken from figure 4.1) into a unit tree.

```
mymodel/
```

[2]Matching, which is also called *semi-unification*, is a concept borrowed from unification theory.

```
layer1/
    A/
        A.java
        A.html
layer2/
    A/
        A.java
        A.html
    B/
        B.java
        B.html
```

When loading a model, Probe traverses the model directory tree and tries to match the category patterns with file names[3]. A successful match results in the creation of a unit that is based on the matching file. Files that do not match are ignored. In our example, directory `mymodel/` matches category `Model`, directory `mymodel/layer1/` matches category `Layer` and file `mymodel/layer1/A.java` matches category `Code`—in that order, because Probe only tries matching the children if the parent matched. The roles for the units produced from these files are derived from the file names as stated in the schema (what exactly happens here is explained later). They are *mymodel*, *layer1* and *code*, respectively.

We briefly mention two features of the schema (consult appendix A for a complete reference on schemas): By specifying the fully qualified name of a Java class in the schema, one can provide a custom implementation for a category of units (that defines, for example, how to compose or display new kinds of artifacts) and for *operations* that are invoked on units through a context menu. The following fragment from a schema file shows how this is done; the names of the classes implementing category `Code` and operation `OpGenerate` are underlined.

```
<artifact category="Code" file="*.java" role="code"
 class="genborg.artifact.Code">
    <menu>
        <operation menuName="Generate code"
         class="genborg.operation.OpGenerateCode"/>
    </menu>
</artifact>
```

Groups, a mechanism for putting files in the same directory into separate collectives without changing the file structure, are also explained in appendix A.

4.2 Properties

Probe has a generic mechanism for specifying *properties* that describe certain characteristics of the units that are created by a category. Role, file name and class of a unit are examples of properties. A unit stores its properties as a set of key-value bindings where each key is unique per unit. It inherits all properties from its ancestors in the unit tree and can either override existing properties (*shade* them for its descendants) or define new ones. Because the role property of a unit needs to be visible in the descendants, its key is the name of the unit's category; choosing the same key `role` for all units would mean that children always override (and hide) their parent's value of this property. All other property names are obvious, like `file`, `class` etc. Until now, we've always used the non-generic way of assigning values to properties, through attributes of the `directory` and `artifact` tags. Look at the following XML code which is taken verbatim from the last schema example:

```
<directory category="Layer" file="*" role="*">
```

[3]Additionally, file patterns of directory categories only match the names of directories.

The above fragment assigns the properties `file` and `role` for every unit produced by category `Layer`, but uses the XML attributes `file` and `role` which are syntactic sugar for generic assigment. We can achieve the same result generically:

```
<directory category="Layer">
    <property name="file" value="*"/>
    <property name="Layer" value="*"/> <!-- Role assigned here -->
```

The definition of property `file` above makes it obvious that we never specify concrete property values in the schema but rather templates that *instantiate* concrete values. A template is a string containing a sequence of variables and literal (constant) text. A variable whose name is `varName` is inserted as `$varName$` into a template. The `*` wildcard used in the definition of `file` is also a variable and syntactic sugar for `$WILDCARD$`.

A template serves two purposes in a schema:

1. It can be used for output. If we have just created a new unit u, a template t instantiates a text string that can be assigned to one of u's properties. Instantiation depends on t and u (or rather, its properties) and works as follows. A variable in t whose name is equal to that of a (potentially inherited) property of u is *bound* to the value of that property. All other variables in t are called *free*. A template that contains only literals and bound variables can be instantiated. An instance $\text{inst}(t, u)$ of t is the result of replacing every variable with the value it is bound to (literals are not changed).

   Take the example of a unit u with the properties { `flower` $\mapsto$ `"rose"`, `bird` $\mapsto$ `"lark"` }; u has no inherited properties. The template string `"There is a $bird$ on this $tree$"` contains variable `bird` that is bound to the value `"lark"` of property `bird`. Variable `tree` is free. Before and after variable `bird`, there is literal text. Template t = `"A $bird$ does not eat a $flower$."` only has bound variables and can therefore be instantiated to the string $\text{inst}(t, u)$ = `"A lark does not eat a rose."`.

2. Every template t can be used for input, as well. In that case, it functions as a pattern that is matched against a string s. The matching algorithm is defined as follows: If we can create properties in u so that all variables in t are bound and $\text{inst}(t, u)$ is equal to s, the match succeeds. Otherwise, it fails. We eliminate ambiguities by demanding that there cannot be two adjacent variables in t and that every literal has to appear in $\text{inst}(t, u)$ as early as possible. The properties of u that were created during matching are called *match-created*. Let's assume the same unit u as in the above examples to illustrate matching:

Template t	String s	Does t match s?
`"Let's go $location$"`	`"Let's go home"`	Yes, we have to match-create one new property for u: `location` $\mapsto$ `"home"`
`"It's still a $flower$"`	`"It's still a lily"`	No, the value of property `flower` is not equal to `"lily"`
`"Hello $bird$"`	`"Goodbye lark"`	No, the literal text doesn't match

So we see that property `file` and property `Layer` in the last schema use the same template string `"*"`, but the former for input and the latter for output. We can now rewrite this schema so that it enforces the constraint that the base name of Java and HTML files be the same as the role of the their parent mixin.

```
<schema>
    <directory category="Model" file="*" role="*">
        <directory category="Layer" file="*" role="*">
            <directory category="Mixin" file="*" role="*">
                <artifact category="Code" file="$Mixin$.java" role="code">
```

```
                </artifact>
                <artifact category="Doc" file="$Mixin$.html" role="doc">
                </artifact>
            </directory>
        </directory>
    </directory>
</schema>
```

4.3 Types

Types in Probe are a simple, syntactic way of ensuring that a function can be applied to a constant. The type of a function f is $t \rightarrow u$, that of a constant c is $\rightarrow v$ (or short, v), where t, u and v are alphanumeric text strings. f can be composed with c if t equals v. The type of $f \circ c$ is then u. Types are specified in the schema through the `type` property. To demonstrate the use of types, we rename the mixin directories of `mymodel/` so that they indicate whether the corresponding unit should be a function or a constant. The file structure is as follows.

```
mymodel/
    layer1/
        constA/
            A.java
            A.html
    layer2/
        funcA/
            A.java
            A.html
        constB/
            B.java
            B.html
```

Note that we want the new directory structure to produce the same model as before. Therefore, even though we've changed the name of, for example, mixin directory `A/` to be `constA/`, the role of its unit should still be A. The prefix of the directory name is just a hint of how to type the unit. We can now type mixins automatically if we change the schema to use more complicated matching patterns.

```
<schema>
    <directory category="Model">

        <directory category="Layer">

            <directory category="ConstMixin" file="const*" role="*">
                <property name="type" value="$ConstMixin$"/>
                <artifact category="Code" file="$ConstMixin$.java" role="code">
                    <property name="type" value="code"/>
                </artifact>
                <artifact category="Doc" file="$ConstMixin$.html" role="doc">
                    <property name="type" value="doc"/>
                </artifact>
            </directory>

            <directory category="FuncMixin" file="func*" role="*">
                <property name="type" value="$FuncMixin$->$FuncMixin$"/>
                <artifact category="Code" file="$FuncMixin$.java" role="code">
                    <property name="type" value="code->code"/>
                </artifact>
```

```
                    <artifact category="Doc" file="$FuncMixin$.html" role="doc">
                        <property name="type" value="doc->doc"/>
                    </artifact>
                </directory>
            </directory>
        </directory>
    </schema>
```

If no type is specified explicitly, the default is `type="role->role"`. Thus, the type of *layer1* is *layer1* → *layer1* and the type of *layer2* is *layer2* → *layer2*. The trick used in this schema is to have two different categories, `ConstMixin` and `FuncMixin`, for mixin classes, that instantiate correctly typed units depending on the prefix of the directory name. Only the suffix of the directory name is used for the role.

4.4 Property Files

Property files allow one to override the property defaults instantiated by the schema. Accordingly, the value part of a property specification is a literal string in the property file and a template in the schema. In order to define properties for a collective, one puts an XML file named `properties.xml` inside the collective's directory. A property file can additionally define properties for any unit of the collective, however deeply buried inside. As an example, we want to change the type of *layer1* so that *layer2* can be applied to it. A quick (and somewhat dirty) way of doing this is by creating a property file `mymodel/layer1/properties.xml` that assigns *layer1* the type *layer2*. The content of `properties.xml` is:

```
<collective>
    <property name="type" value="layer2"/>
</collective>
```

To demonstrate how property files can also specify properties for primitive units (which are not defined by a directory and therefore can't have their own property file), we assign to a hypothetical `color` property of the unit whose *role path* is *layer2/A/code*. We achieve this by putting a property file `mymodel/layer2/constA/properties.xml` into the directory of its parent unit, the mixin whose role is *layer2/A*[4]. The property file contains the usual property definition, but inside a `unit` tag that indicates that we want this definition to apply to a subunit of A, whereas top-level property definitions apply to A:

```
<collective>
    <unit file="A.java">
        <property name="color" value="blue"/>
    </unit>
</collective>
```

Note that the subunit is identified by file name which makes it easier for external tools to create property files.

4.5 Project Directory

The parts of Probe we have introduced can read a model's content from disk into RAM. But we also want to create and save equations. We found that the easiest way to make the result of an equation

[4]Watch out for the difference between roles/role paths and files/file paths! The unit that plays role *layer2/A* is based on the directory `mymodel/layer2/constA/`.

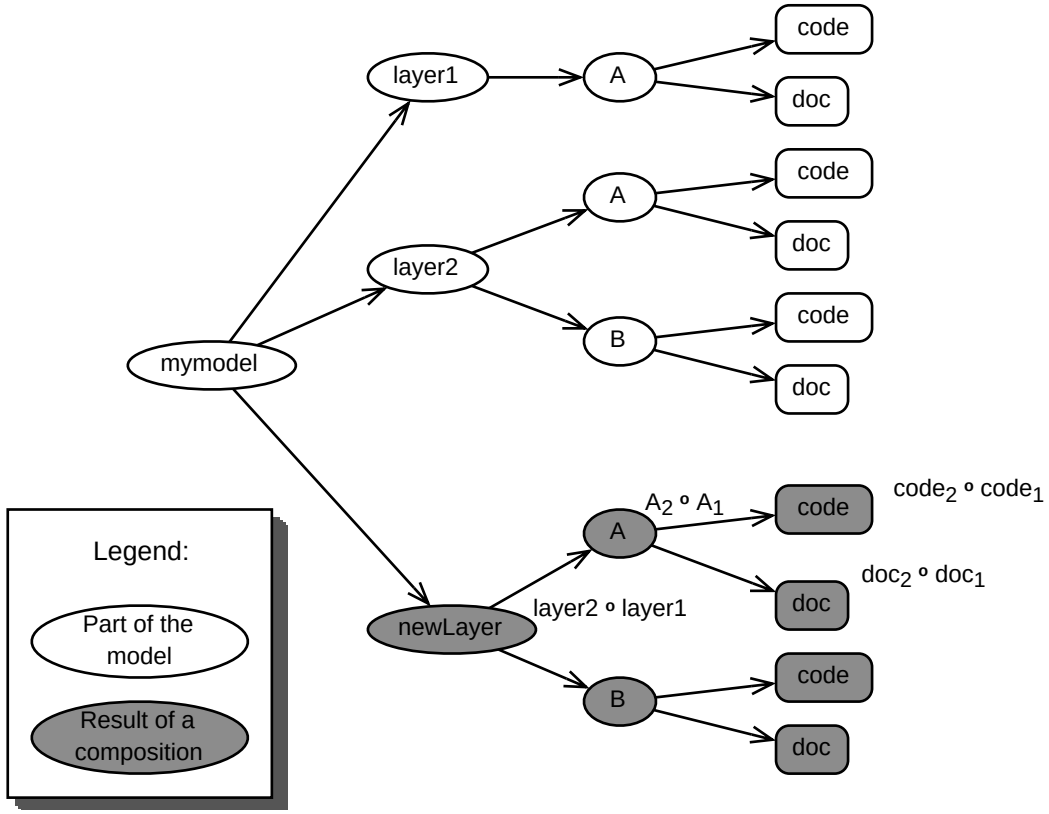


Figure 4.3: The result of the equation $newLayer = layer2 \circ layer1$ is added back into the unit tree. Every node contains its role, nodes that resulted from the composition are annotated with their equations where necessary. Indices in the annotation indicate the source of a constituent.

browsable is to put it back into the unit tree. It becomes a member of the same collective as the units that have been used in its creation and has to be assigned a role that is unique in that collective (figure 4.3). Essentially, this generalizes the notion of a collective which can now contain functions and expressions (composed from these functions or even other equations). Probe keeps track of the source (a composition or the model) of each unit by tagging it with an *owner* object; equations have different owners from uncomposed units. The owner of an equation takes care of saving it (see below).

Because the same model might be shared by different projects, Probe creates one *project directory* per project and saves equations inside it. A project directory has the same structure as a model directory, including property files which are automatically created if the user wants to override any of the computed values. But it does not contain a schema file. Instead, there is a *manifest*, an XML file that is stored in the root of the project directory as `manifest.xml`. The manifest points to the model directory of the project and also records all the compositions that have been made. Every time Probe loads a new project into RAM, it first reads its model and then recomputes the recorded equations. That means that while the model is persistently stored on disk, the result of an equation is not. The files of the equation's result are only written to disk when the users issues a menu command. This on-demand approach is slower at project load time, but quicker when creating new equations and makes it unnecessary to save computed properties like role and type. The following XML code is the content of a manifest file that stores two compositions, $newLayer = layer2 \circ layer1$ (line 2–5) and $anotherLayer = layer2 \circ newLayer$ (line 6–9). It also points to the model directory `/data/models/mymodel` and contains the version of the file format, 1.0.0 (both in line 1).

```
1  <manifest modelDir="/data/models/mymodel" version="1.0.0">
2      <composition role="newLayer">
3          <unit rolePath="layer2"/>
4          <unit rolePath="layer1"/>
5      </composition>
6      <composition role="anotherLayer">
7          <unit rolePath="layer2"/>
8          <unit rolePath="newLayer"/>
9      </composition>
10 </manifest>
```

We are now ready to explain why the mapping from file names to roles specified in the schema has to be invertible. In the model of figure 4.3, if the user asks Probe to generate the artifacts of *newLayer*, it has to know where in the project directory it should create the new files. The solution is to first translate each path of roles into a file path. This path is then taken to be relative to the directory of the owner of a unit and turned into an absolute file path. Take the role path *newLayer/A/doc*[5] which refers to the result of composing *layer2/A/doc* with *layer1/A/doc*. It translates to the relative file path `newLayer/A/A.html`[6]. The owner directory of *doc* is the project directory, say `/home/john/demo_proj`. Therefore the final absolute path name is `/home/john/demo_proj/newLayer/A/A.html`.

4.6 The Unit Tree as a Relation

Apart from the obvious presentation as a tree, Probe also displays a model as a relation. This solves several problems related to browsing that we will motivate first. Let's take another look at the example model of section 2.2 (repeated in table 4.1). The advantage of showing this model to the user as a tree (figure 4.4) is that it corresponds to how the model is saved in RAM and on disk and that navigating a tree graphically is well understood. Java's GUI library, Swing, even provides a standard control for displaying trees. Traversing the tree, it is easy to find out which mixin classes belong to *Layer2*; we are looking at the rows of table 4.1 (1). But what if we want to know what parts make up class *A*, a

[5] By convention, role paths do not include the root role.

[6] Here we are facing the limits of the legacy type system and have to use the older schema from page 25, which did not automatically type units based on file names. The auto-typing schema on page 30 would incorrectly create a file path `newLayer/funcA/A.html` that does not correspond to its constant type. This is due to the way Probe propagates categories in a composition.

A_1	B_1	C_1	Layer1
A_2	B_2		Layer2
	B_3	C_3	Layer3
A_4	B_4	C_4	Layer4
Class A	Class B	Class C	

Table 4.1: The example from section 2.2 as a GenBorg model. Cells represent mixin classes, rows layers and columns complete classes. The numeric indices are only used to indicate the source layer and do not appear in the model.

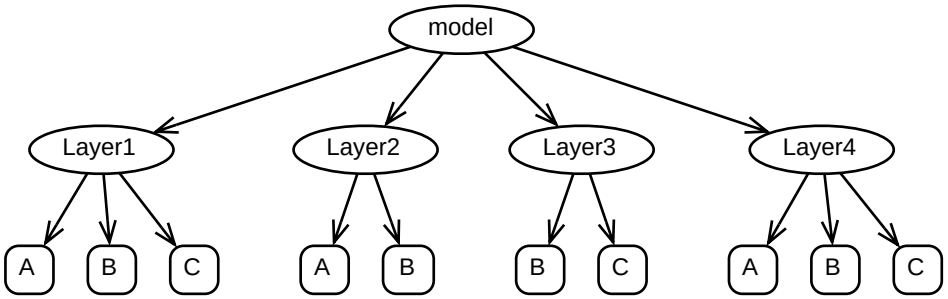


Figure 4.4: The model of table 4.1 displayed as a tree.

column of the table (2)? It is very difficult to browse the tree to get an answer. Probe's solution is to transform the tree into a database relation. Browsing a model then means to query the relation in a database language. The following rules turn the model tree into a relation:

1. The relation holds all units of the model with the exception of the root unit. The key of a record is the path (excluding root) to the unit as an array of roles. When we display a record, we omit the unit field.

2. If a schema has n levels, the maximum length of a path is $n-1$. A key that is shorter than $n-1$ contains *null roles* to fill the empty columns. Null roles are displayed as <widgets> around the name of the unit's category (e. g., `<Layer>`). If the unit is primitive, null roles are inserted *before* the last role in its path[7]. Null roles in the paths of collectives are inserted at the end.

3. The roles in the i-th column are all from level i of the unit tree (the level of root is 0). Accordingly, the title of that column is "Level i". If there is only one category at level i in the schema tree, we rename the column to be the name of that category (we also call the last column "Artifact").

Applying these rules to the example model from figure 4.4 produces the relation

[7]This has the effect of right-justifying the roles of leaves, uniting the roles of artifact units in the last column.

$Rel :=$

Layer	Artifact
Layer1	<Layer>
Layer1	A
Layer1	B
Layer1	C
Layer2	<Layer>
Layer2	A
Layer2	B
Layer3	<Layer>
Layer3	B
Layer3	C
Layer4	<Layer>
Layer4	A
Layer4	B
Layer4	C

Now we can answer both of the above questions, instead of just the first one: Listing the content of *Layer2* is achieved by the SQL statement

```
select * from Rel where Layer="Layer2"
```
(1)

The result of this query is

Layer	Artifact
Layer2	<Layer>
Layer2	A
Layer2	B

It lists all paths that start with role *Layer2*: Row 1 refers to *Layer2* itself whose path is too short and filled with a null role in column *Artifact*. Row 2 holds the path to artifact A, a child of *Layer2*, and row 3 lists artifact B's path. The following statement retrieves all the mixins that contain a class A.

```
select * from Rel where Artifact="A"
```
(2)

It returns

Layer	Artifact
Layer1	A
Layer2	A
Layer4	A

For visual browsing, we want to get rid of SQL and represent a query as a tuple[8]. Any query

```
select * from Rel where key_i1=value_i1 and ...  and key_im=value_im
```

(where `key_j` is the title of column `j` , `value_j` is any of the values in that column and $1 \leq i_k \leq n-1$) can be converted to a tuple $t \in (\pi_1(Rel) \cup \{*\}) \times \ldots \times (\pi_{n-1}(Rel) \cup \{*\})$ as follows. The projection function $\pi_j(s)$ returns the set of values of column j of relation s.

$$t_j := \begin{cases} value_j & : \text{ if } \quad j \in \{i_1, \ldots, i_m\} \\ * & : \text{ otherwise} \end{cases}$$

The tuples for example (1) and (2) are `(Layer, *)` and `(*, A)`. The graphical user interface for specifying t is a set of $n-1$ *select lists* L_j that display $\pi_j(Rel)$. That way, select list L_j presents all possible values for component t_j of the query tuple. Our example model has the following two select lists:

[8] reminiscent of the query language *Query by Example* from IBM [Zloof, 1975].

Layer	Artifact
Layer1	A
Layer2	B
Layer3	C
Layer4	<Layer>

Selecting value $value_j$ in list L_j means that $t_j = value_j$, selecting nothing indicates that $t_j = *$. When one selects strictly from left to right and does not skip a list (i. e., zero or more lists on the left are selected, the remaining lists are unselected), the user experience is similar to browsing the file system in Mac OS X (née NeXTStep) or navigating the class hierarchy in SmallTalk.

Chapter 5

Tutorial

This chapter is a tutorial for Probe. We'll use the model *HTMLCompose* to demonstrate the features of this program. *HTMLCompose* is actually used by Probe to compose HTML files.

5.1 Model Directory

Before we start, we want to examine the model directory of `HTMLCompose`. It consists of the following files:

```
HTMLCompose/
|-- Begin
|   |-- Action.java
|   |-- Constants.java
|   |-- Decl.java
|   |-- Doc.html
|   |-- Error.java
|   |-- HTMLfile.java
|   |-- MLObject.java
|   |-- Main.java
|   |-- Pair.java
|   |-- Parse.java
|   |-- ProgramText.java
|   |-- StringWrapper.java
|   |-- blocks.gif
|   |-- comp1.gif
|   |-- comp2.gif
|   `-- properties.xml
|-- Call
|   |-- Constants.java
|   |-- Doc.html
|   |-- Parse.java
|   |-- call.java
|   `-- properties.xml
|-- Copy
|   |-- Constants.java
|   |-- Copy.java
|   |-- CopyFile.java
|   |-- Doc.html
|   |-- FileName.java
|   |-- HTMLfile.java
|   |-- Parse.java
|   |-- StringPair.java
|   `-- properties.xml
|-- Extend
|   |-- Constants.java
|   |-- Doc.html
|   |-- Extend.java
|   |-- Parse.java
|   `-- properties.xml
|-- Method
|   |-- Constants.java
|   |-- Doc.html
|   |-- Pair.java
|   `-- Parse.java
`-- schema.xml
```

The model's schema file `HTMLCompose/schema.xml` is

```
1   <schema>
2
3       <directory category="Model">
4
5           <directory category="Layer">
```

```
 6
 7               <property name="type" value="Layer->Layer"/>
 8
 9               <menu>
10                   <operation
11                       menuName="Compose layers"
12                       class="genborg.operation.OpCompose"/>
13                   <operation
14                       menuName="Delete code"
15                       class="genborg.operation.OpDeleteCode"/>
16                   <operation
17                       menuName="Delete documentation"
18                       class="genborg.operation.OpDeleteDocumentation"/>
19                   <operation
20                       menuName="Generate code"
21                       class="genborg.operation.OpGenerateCode"/>
22                   <operation
23                       menuName="Generate documentation"
24                       class="genborg.operation.OpGenerateDocumentation"/>
25               </menu>
26
27               <group category="Code">
28                   <artifact
29                       category="JavaCode"
30                       file="*.java"
31                       role="*" groupRole="Code"
32                       class="genborg.artifact.Code">
33
34                       <menu>
35                           <operation
36                               menuName="Generate code"
37                               class="genborg.operation.OpGenerateCode"/>
38                       </menu>
39                   </artifact>
40               </group>
41
42               <group category="Doc">
43                   <artifact
44                       category="HTML"
45                       file="*.html"
46                       role="*"
47                       groupRole="Doc"
48                       class="genborg.artifact.Documentation">
49
50                       <menu>
51                           <operation
52                               menuName="Generate documentation"
53                               class="genborg.operation.OpGenerateDocumentation"/>
54                       </menu>
55                   </artifact>
56               </group>
57           </directory>
58       </directory>
59   </schema>
```

Let's go through the categories.

- `Model` (line 3): The category of the root collective.

- `Layer` (line 5) overrides the `type` default and defines several operations a user can perform on a layer (line 9–25). Each operation is implemented by a Java class, for example, class `genborg.operation.OpCompose` implements an operation that provides a graphical user interface for composing layers. After the user has specified what layers he wants to compose, class `OpCompose` internally invokes methods on unit objects that perform the composition. Operations are implemented by overriding a method of an abstract superclass (see section D.2).

- `Code` (line 27) is a new kind of category, a *group*, which behaves like a *virtual directory*: Units of files that match its direct subcategories are put into collectives that do not correspond to a file or directory. So files in the same directory will be in separate collectives, as if one created directories where these files now are and moved them there. Consult appendix A for details.

- `JavaCode` (line 28): Artifacts in this category are implemented through the Java class `genborg.artifact.Code` (line 32). The purpose of this class is to implement composition for `JavaCode` artifacts and to tell Probe how to display them. If no class is given, Probe uses the default implementations `genborg.base.Artifact` for artifacts and `genborg.base.Collective` for collectives. These classes take care of most of the algebraic book-keeping (creation and typing of composed units etc.), but do not modify files or display units. Custom implementations subclass `Artifact` and `Collective` to add new functionality (see section D.1), such as transformation of files or a special way of presenting a unit's data. External tools are integrated into Probe by wrapping their invocation in a custom unit implementation.

  There is one operation that can be performed on `JavaCode`, "Generate code" (line 35).

- Categories `Doc` and `HTML` work analogously to `Code` and `JavaCode`.

5.2 Files in the Model Directory

The property files in layer directories explicitly assign types where the defaults given in the schema don't apply. For example, `Begin/properties.xml` specifies that layer *Begin* (line 2) is a constant and also overrides type defaults as constants for its subunits:

```
1  <collective>
2     <property name="type" value="Layer"/>
3     <unit file="Action.java">
4        <property name="type" value="Action"/>
5     </unit>
6     <unit file="Constants.java">
7        <property name="type" value="Constants"/>
8     </unit>
9     <unit file="Decl.java">
10       <property name="type" value="Decl"/>
11    </unit>
12    [several entries omitted here]
13 </collective>
```

Part of the elegance of the current GenBorg implementation stems from the fact that the source code files in a model directory contain pure Java. Each file under Probe defines a mixin class[1]. Therefore, the only addition to Java that is needed is a parameter for the superclass, as this is the one distinguishing characteristic between normal classes and mixins. This parameter is only filled in when the layer of a mixin class is used in a composition. It is determined by what (potentially accumulated) collective is refined and by what role the mixin class plays. The former specifies a set of classes that can be refined, the latter selects a member from this set (which is the final value of the parameter). Because the equation is defined externally, all we have to do is to state the role of a class. Probe takes

[1] We generalize a traditional Java class, which is also called a *base class*, to be a mixin class whose "superclass" parameter is null (empty).

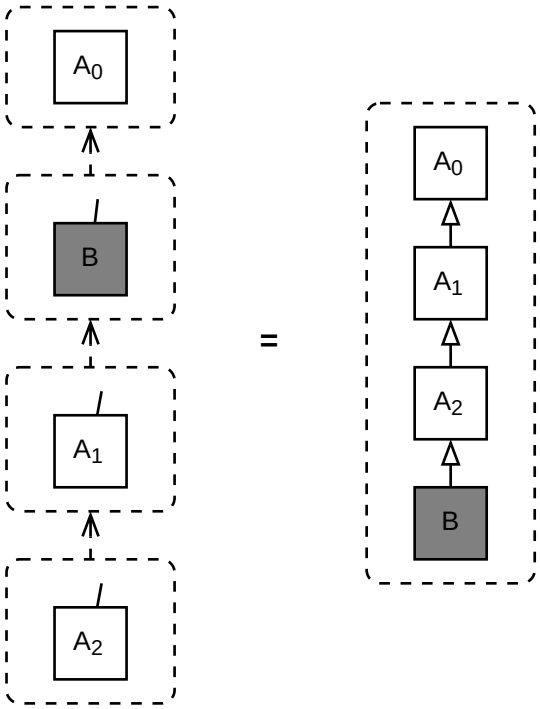


Figure 5.1: Class B extends class A_0, class A_1 refines A_0, and then A_2 refines A_1 (left). Therefore, B is inserted into the inheritance chain after the *most refined* version of A and A_1 directly after A_0 (right).

the class name to be the role of the class.[2] Thus, every class is a mixin that refines other classes with the same name, if there are any, and a base class, otherwise.

Note that there is a difference between class A_1 refining class A_0 and class B extending A_0. Subclassing, the latter case, always applies to the *most refined* version of a class (figure 5.1). So if we view the final inheritance chain, there might be a couple of other classes between A_0 and B that have been added by subsequent layers (as refinements of A_0). The only thing that distinguishes extension and refinement of A_0 is therefore the point of insertion of a class into the inheritance chain: The former inserts A_1 "right now", the latter waits with inserting B until all refinements for A_0 have been added. The syntax for extending a class is identical to Java, even if the superclass (or rather, "superrole") is in another layer.

5.3 Composition of Non-Code Artifacts

The idea behind composing the non-code artifact *HTML file* is that if you can transfer the concepts *method* and *overriding* from object-oriented programming to HTML, then it is easy to express composition. We define *Methods* in an HTML file to be labeled sections of HTML code that have been bracketed by a pair of `<method name="...">` and `</method>` tags. The method tags themselves are ignored[3] by HTML browsers, but they play an important role in extending an HTML file. All content of an extension file is appended to the data that has been accumulated so far, the exception being methods that *override* other methods (this happens when the name labels are equal). The overriding method is moved out of its surroundings and replaces the content it overrides (figure 5.2). To reuse an overridden method, a `<super>` tag is placed inside the body of the overriding method. A copy of the content of the former replaces the `<super>` tag in a composition. The extension file can also define

[2]This is the canonical way of defining roles in the schema. In general, the only requirement is that there be a one to one correspondence between roles and class names.

[3]Other, less obtrusive, ways of marking methods can easily be swapped for this mechanism: Hiding the markers inside comments, using attributes for the standard `<div>` tags, etc.

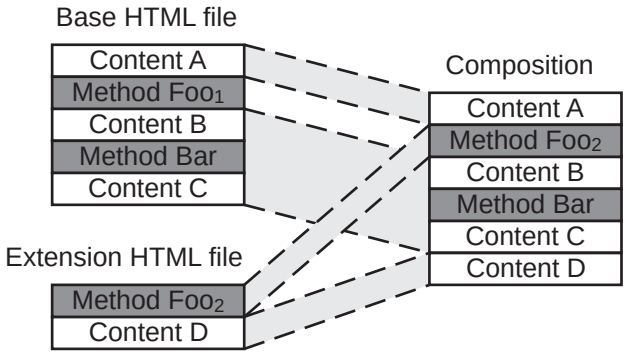


Figure 5.2: Composing two HTML files. The extension file overrides method `Foo`$_1$ with `Foo`$_2$ (indices only indicate the location of a method and do not appear in the method names). Otherwise, the two files are just concatenated.

new methods that can be overridden by subsequent extensions.

This file format specializes in documenting method definitions in Java classes. Therefore, composition of these HTML files mimics the way subclasses would be constructed from Java source code if one had to copy the content of a superclass instead of referencing it: The source code of a superclass A and subclass B would be concatenated and any overriding method of B would be moved to replace the text of the overridden method in A. The following two files demonstrate composition of HTML[4]. The first file is a base file.

```
<!-- Base file -->
The composition is capable of
<method name="capabilities">
singing
</method>
```

The second file refines the base file.

```
<!-- Extension file -->
<method name="capabilities">
<super>
and dancing.
</method>
And it also <i>looks good</i> while doing it.
```

The composition of these two files looks like this:

```
<!-- Base file -->
The composition is capable of
<method name="capabilities">
singing
and dancing.
</method>
<!-- Extension file -->
And it also <i>looks good</i> while doing it.
```

We still have not explained the GIF files in `HTMLCompose/Begin/`: `blocks.gif`, `comp1.gif` and `comp2.gif`. These files are referenced by `Doc.html`. They correctly do not appear in the schema[5] (the idea is that an HTML file encapsulates its content), but are copied along when composing HTML.

[4]There is a prototype tool, called *XC*, that accomplished much of what is said in these examples. The primary difference is that XC distinguishes a method declaration from a method call, which the examples given here don't.

[5]XC also has the capability of copying image files directly when composing `.html` files. Alternatively, we could have

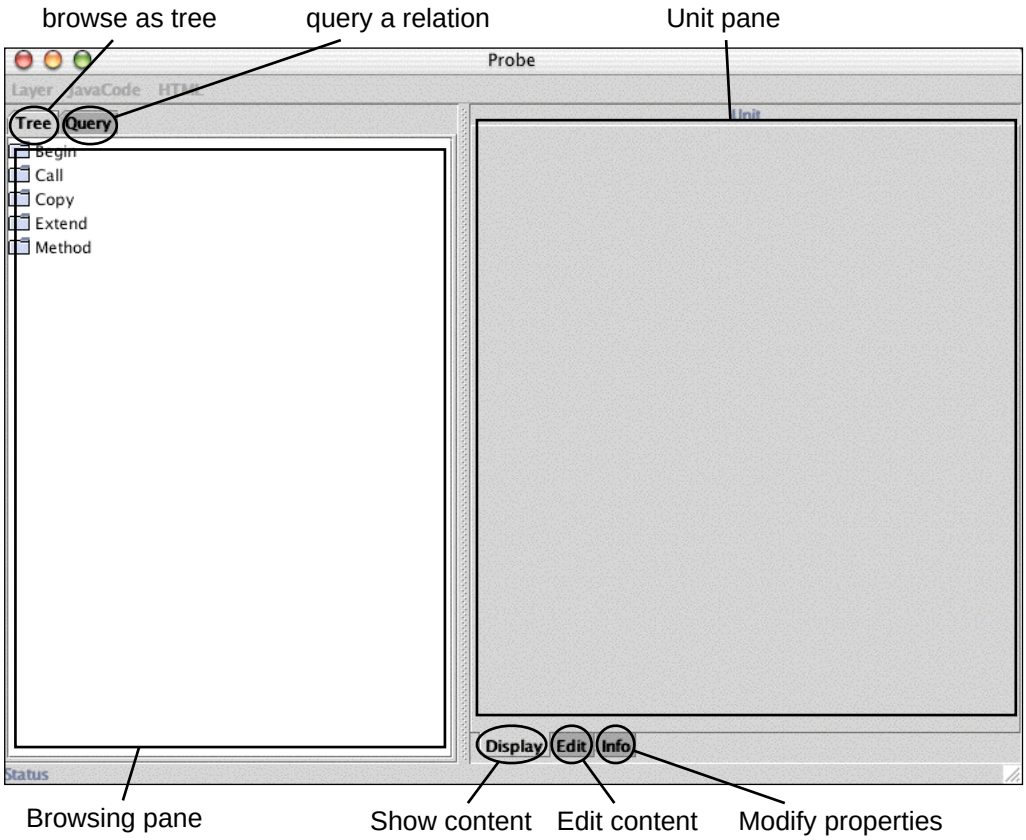


Figure 5.3: Probe starts up and shows the model

5.4 Starting Probe

Probe is packaged as a Java archive (JAR) file that contains everything that's needed for running the program. If one starts it without any arguments, Probe gives a short usage description. The following lines show an interaction with the application in the Unix shell `tcsh`.

```
[~/probe] % java -jar Probe1.0.jar
Usage:
Load project:       java genborg.gui.Viewer                  projDir/
Create project dir: java genborg.gui.Viewer -model modelDir/ projDir/
```

This implies that, to work with Probe, we initially need to create a project directory and specify a model directory. In subsequent sessions, providing the program with a project directory is enough, as it contains a reference to the model directory. The following shell command creates a new project directory `proj` whose model directory is `HTMLCompose` and starts Probe.

```
[~/probe] % java -jar Probe1.0.jar -model HTMLCompose/ proj/
```

Probe's window comes up (figure 5.3). The window is split in half: The left side is used for browsing and offers two methods for selecting units (as indicated by the tabs on top): A tree view of model and project (tab `Tree`) or a relational view that can be queried as explained in section 4.6 (tab `Query`). The right side displays a selected unit in three different modes: one can view the content of a unit (tab `Display`), edit it (tab `Edit`) or modify the properties (tab `Info`).

modified the schema to admit `.gif` files as an artifact type, and have such artifact instances "copied" for us during the composition of `.html` files. Note, however, that we would not define composition (refinement) operators on `.gif` files.

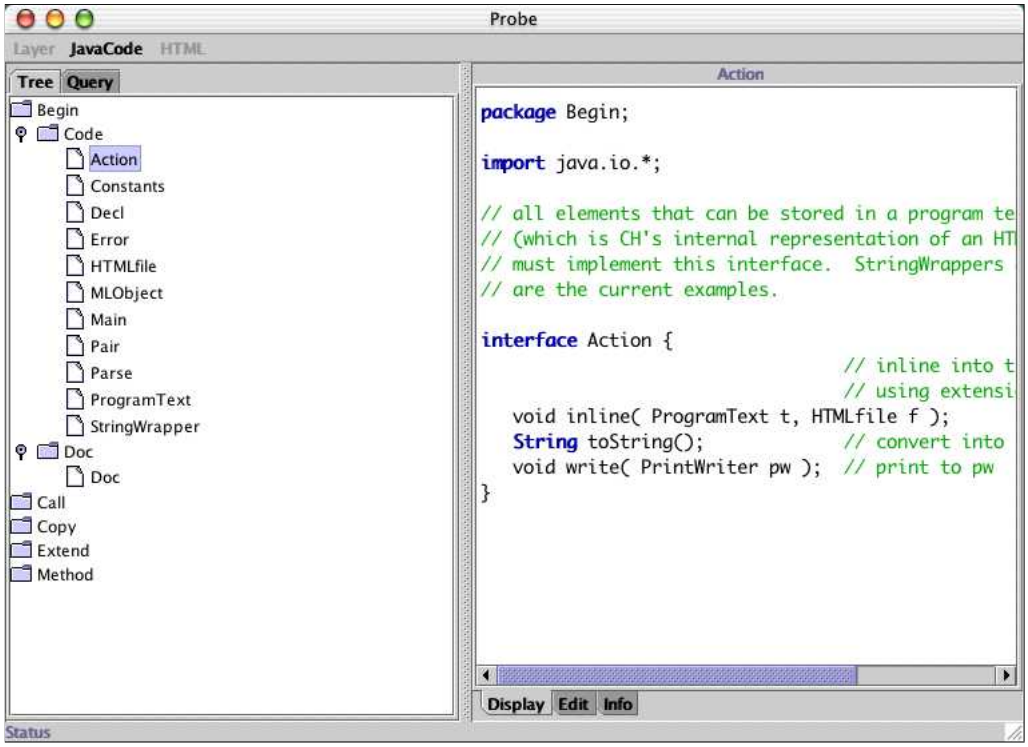


Figure 5.4: Selecting the *Action* unit that contains Java code

5.5 Tree Browsing

The `Tree` displays the unit tree in its natural format, as a hierarchy. A first glance shows us that all the subdirectories of the model directory are there, as layers. Exploring layer *Begin* confirms that the Java and HTML files in its directory have been turned into units, as well. If we click *Begin/Code/Action*, the unit pane on the right displays its Java code as a syntax-highlighted listing (figure 5.4). The content of *Begin/Doc/Doc* is shown as rendered HTML (figure 5.5). Let's try the other ways of viewing a unit's content. Click on tab `Edit` and an editable field with the HTML source in plain text comes up. If we modify the source, the `save` button becomes active and we can save the changes (figure 5.6). Clicking on tab `Info` makes it possible to view the properties of unit *Doc*. It only makes sense to edit property `type` and once we do that, the buttons `Reset` and `Save` are enabled, permitting us to either discard or commit what we have changed (figure 5.7).

There are two ways of invoking the operations defined in the schema on the currently selected unit: Either as a popup (context) menu or as a pulldown (menu bar) menu (figure 5.8). To create an equation, we choose operation "Compose layers" for any layer. A dialog box comes up that lets us specify the role (left-hand side of the equation) and the composition (right-hand side of the equation). The left half of the dialog lists all potential operands for the composition, which is displayed in the right half, using the legacy application notation. Clicking on any part of the composition deletes this part. Our equation is $HTMLComposer = Method \circ Extend \circ Copy \circ Call \circ Begin$ (figure 5.9). After we click `OK`, the composed unit is added to the tree under its new role *HTMLComposer*. It shows up in the left half of the Probe window and can be browsed as usual. Testament to *HTMLComposer*'s subunits being different from the other units is the information given for the `Info` tab: It lists both the composition and its parts (figure 5.10). Also, the `Display` tab of, e. g., *HTMLComposer/Code/Parse* only shows the text message `Artifact file "/Users/rauschma/probe/proj/HTMLCompose/HTMLComposer/Parse.java" not yet generated`. The missing artifact file obviously has to be generated before it can be displayed. We invoke operation "Generate code" on layer *HTMLComposer* and can now look at *Parse*'s content.

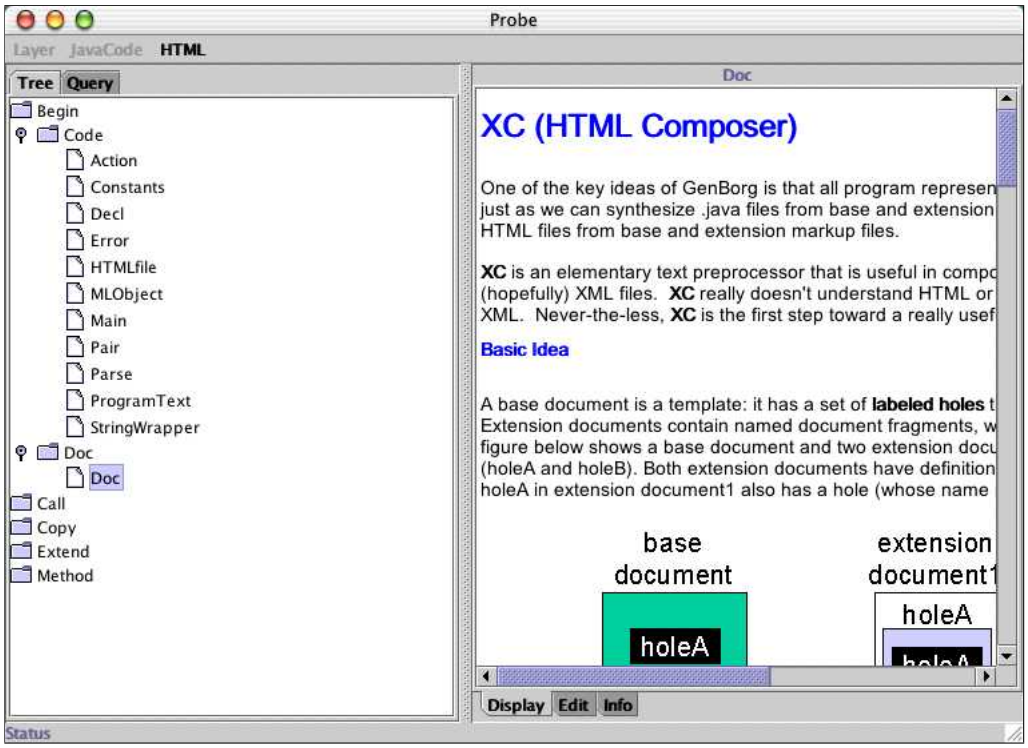


Figure 5.5: Selecting the *Doc* unit that contains HTML code

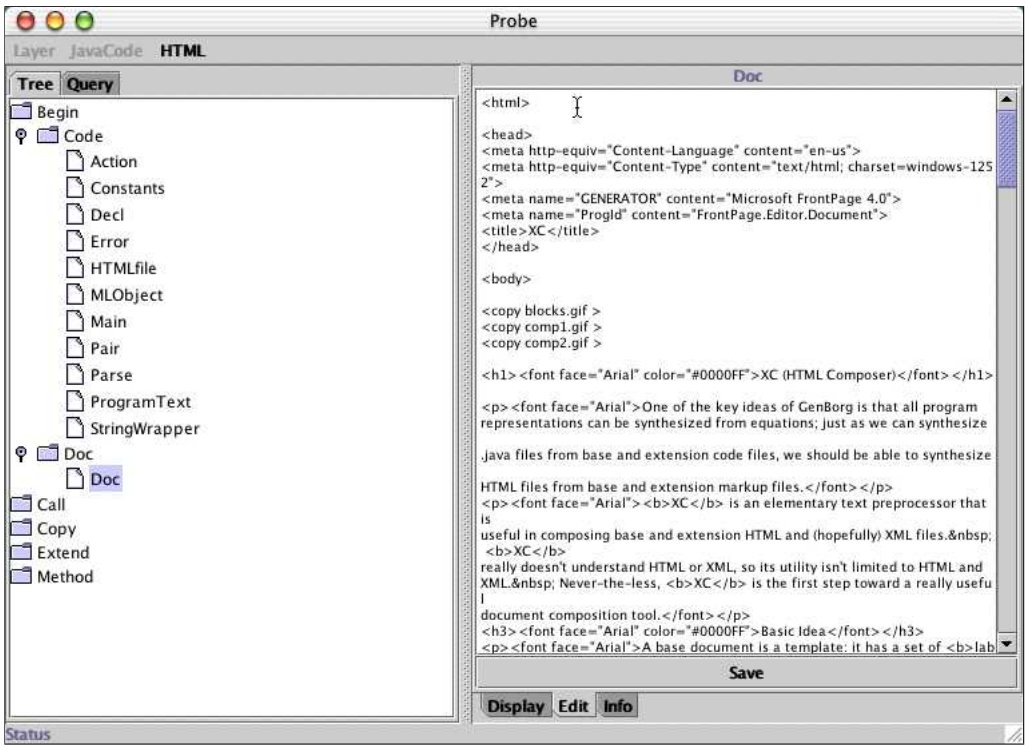


Figure 5.6: The `Edit` tab lets one edit the content of the *Doc* unit

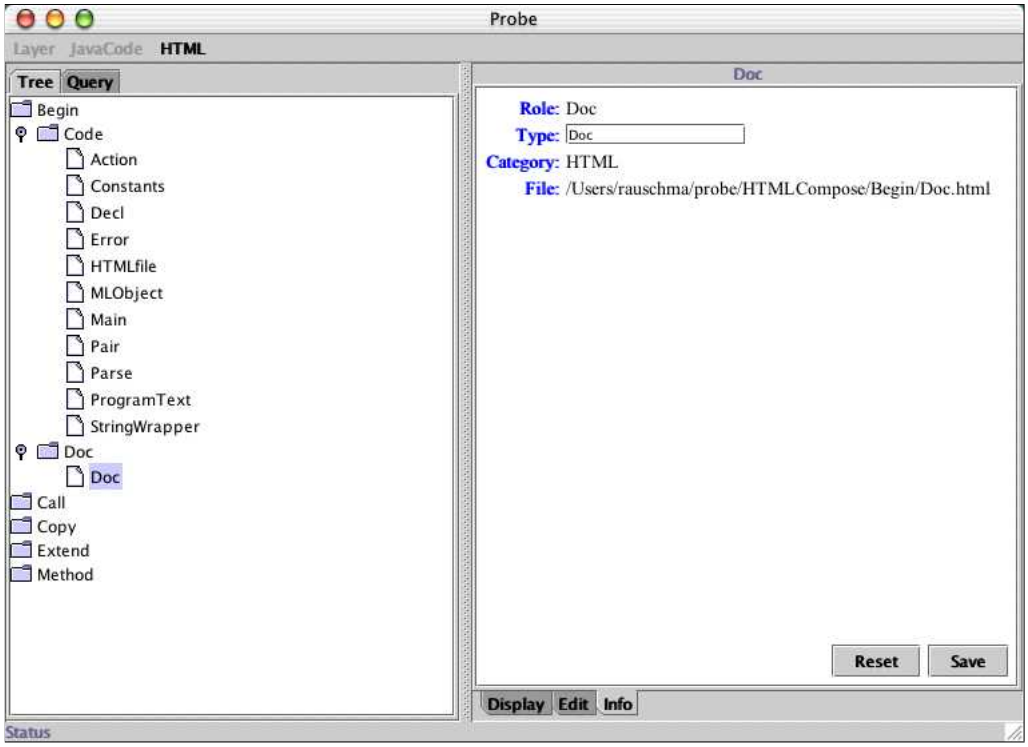


Figure 5.7: The `Info` tab shows *Doc*'s properties

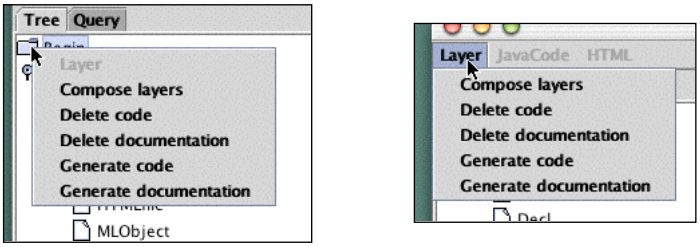


Figure 5.8: There are two ways of accessing a unit's operations: As a popup menu (left) or as a pulldown menu (right).

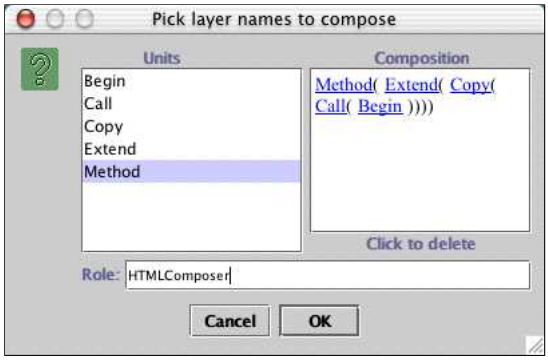


Figure 5.9: Composition dialog for specifying an equation. The left half shows a list of units that is available for composing, the right half displays the current composition. The role text field gives the equation a name.

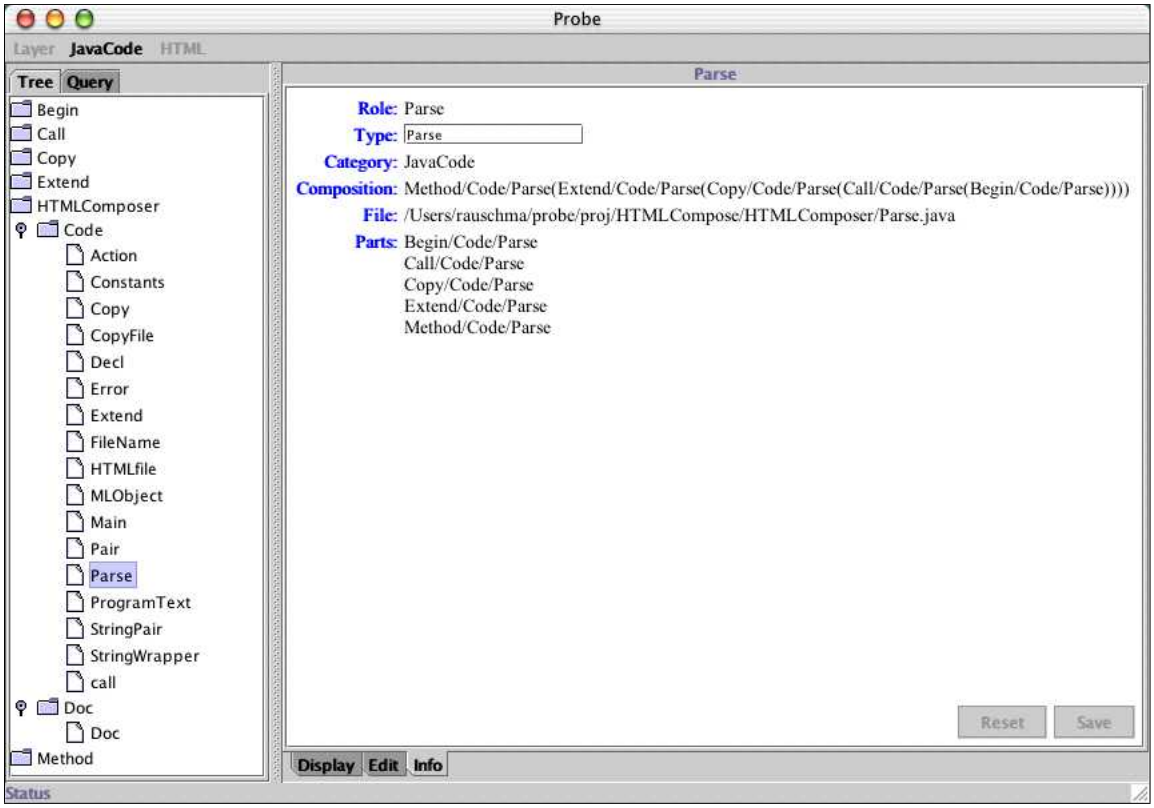


Figure 5.10: The info for *HTMLComposer/Code/Parse* shows that this is a composed unit and lists its parts.

5.6 Query Browsing

The `Query` tab in the top left corner provides us with more sophisticated means for browsing: The unit tree is displayed as a relation and can be queried through a column of *select lists* (figure 5.11). At first, the whole model is displayed in the results. Notice that the unit that was last selected under the `Tree` tab is selected here, too. The selections of the two tabs are always synchronized. The equivalent of the tree gadget is only the right half of the tab, the query result; the select lists narrow the results, but are not used for activating units. Therefore, specifying a query has no effect on what unit is currently displayed. We perform two queries: First, we want to find out what's in layer *Begin* and click on its role in list `Layer`. The query results change and display an answer to our question (figure 5.12). Before we execute the second query, we clear our selection by clicking on the `Deselect all` button (we also could have clicked on the title of the `Layer` select list, which deselects just this one list). Then we click on role `Pair` in list `Artifact` which shows us what layers *Pair* is part of. The titles of the select lists also change to reflect the current query results (figure 5.12).

5.7 Project Directory

Look at the files that have been generated for the new layer *HTMLComposer*. The project directory now contains:

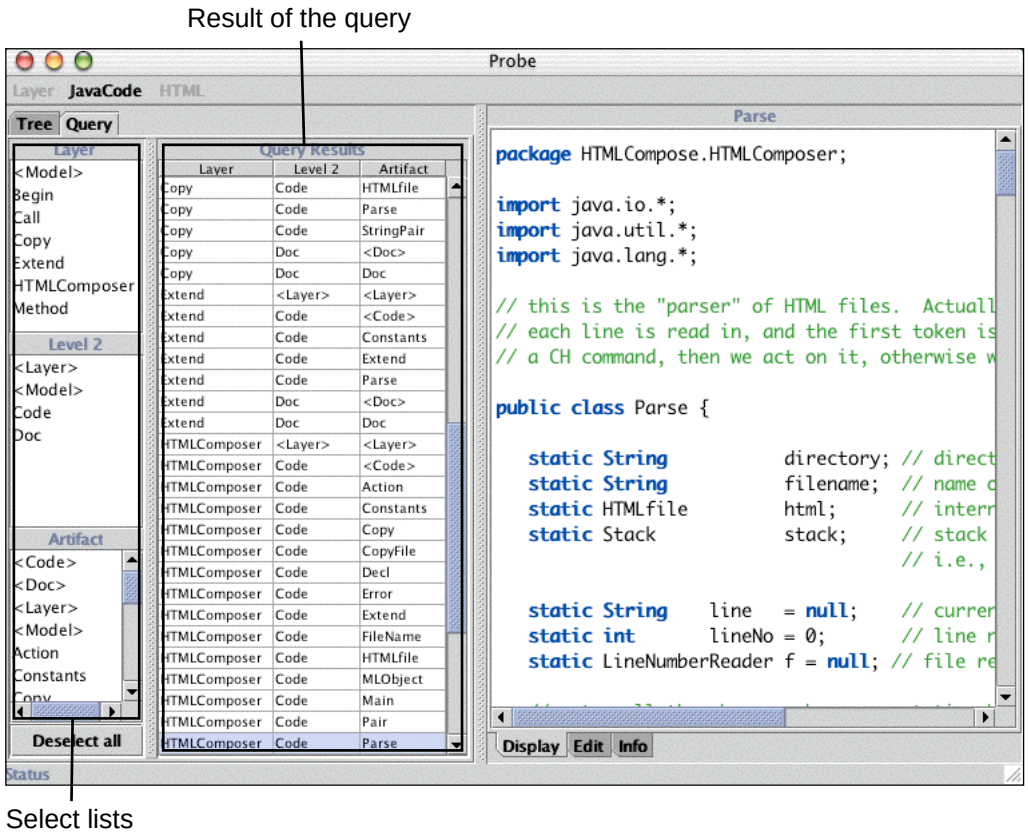


Figure 5.11: The `Query` tab shows the model as a relation that can by queried. The parameters of the query are given in the select lists.

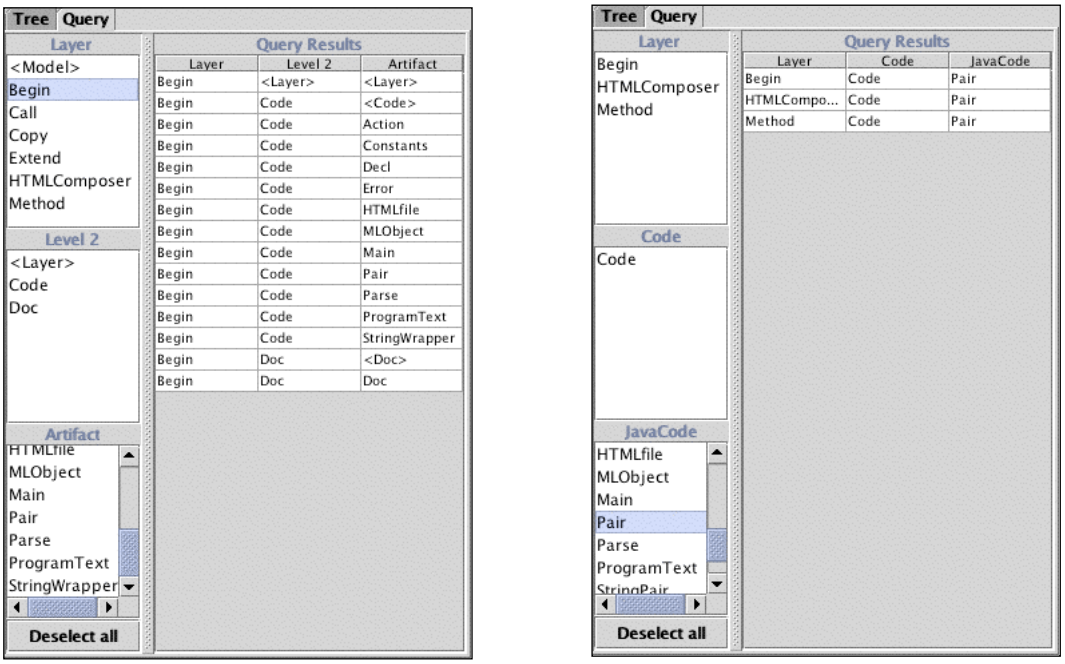


Figure 5.12: Example queries. Left: What are the contents of layer *Begin*? Right: What layers is *Pair* a part of? These queries represented as a tuple (see section 4.6) are: `(Begin, *, *)` for the left and `(*, *, Pair)` for the right query.

```
proj
|-- HTMLCompose
|   `-- HTMLComposer
|       |-- Action.java
|       |-- Constants.java
|       |-- Copy.java
|       |-- CopyFile.java
|       |-- Decl.java
|       |-- Error.java
|       |-- Extend.java
|       |-- FileName.java
|       |-- HTMLfile.java
|       |-- MLObject.java
|       |-- Main.java
|       |-- Pair.java
|       |-- Parse.java
|       |-- ProgramText.java
|       |-- StringPair.java
|       |-- StringWrapper.java
|       `-- call.java
`-- manifest.xml
```

The project directory has one subdirectory for the model *HTMLCompose* and one for the composed layer *HTMLComposer*. Each file in the latter subdirectory is the result of composing one or more units. File `Parse.java`, for instance, depends on units from layers *Begin*, *Call*, *Copy*, *Extend*, *Method*, as shown in figure 5.10. The project manifest file `proj/manifest.xml` records that we have made one composition:

```
<manifest modelDir="/Users/rauschma/probe/HTMLCompose" version="1.0.0">
    <composition role="HTMLComposer">
        <unit rolePath="Method"/>
        <unit rolePath="Extend"/>
        <unit rolePath="Copy"/>
        <unit rolePath="Call"/>
        <unit rolePath="Begin"/>
    </composition>
</manifest>
```

This concludes our tutorial of Probe. By walking through a practical example, we have seen how Probe defines a persistent data structure for a GenBorg model, acts as a *make*-like build tool, and visualizes parameterization and result of the build process.

Chapter 6

Conclusion

6.1 GenBorg

The major achievement of GenBorg compared to other approaches like Hyper/J and AspectJ is that it *has* theoretical underpinnings that explain what a composition of artifacts means. AspectJ looks just at the practical side and Hyper/J's model only explains how and why atomic units of code are grouped. Both programming paradigms also fail to support non-code artifacts. It is briefly mentioned as necessitating future research for Hyper/J in [Ossher and Tarr, 2000] and ignored in AspectJ. Conversely, GenBorg already has a working prototype, Probe, that can compose and generate both code and documentation. Composition of other artifact types, such as JavaSM, an extension of Java that supports state machines [Batory et al., 2000b], is understood—the same mechanisms for generating code must be carried over to non-code artifacts. Implementing GenBorg as Probe has already helped us to considerably simplify the GenBorg algebra. Our work on Probe also brought up ideas about how to improve GenBorg. It currently lacks two capabilities of Hyper/J that should be easy to add:

1. *On-demand remodularization*: Hyper/J allows almost arbitrary cuts through an existing program using *hyperslices*. Several hyperslices can be extracted from the same compiled Java classes and integrated just like hand-written hyperslices. This process is called *on-demand remodularization* and leads to a variety of maintenance and integration problems if you want to evolve two different hyperslices based on the same classes. It also makes composition unnecessarily complicated in Hyper/J. We think insights into the structure of a development project should be reflected in its source code and favor refactoring [Fowler, 1999] by splitting a collective if it becomes apparent that it contains tangled concerns. And refactoring usually is not handled by a programming paradigm, but rather supported by the tools that are based on it. It would make sense, though, to theoretically back parameterized read-only views of collectives by a *projection operator*. The beginnings of such an operator can be seen in the query feature of Probe.

2. *Powerful means of quantification*: Hyper/J's mechanisms for distributing pieces of code inside an existing code base are very complex. Conversely, GenBorg's *composition by role*, while being simple and clean, is less powerful. Take the example of the concern "execution tracing" that can be implemented as a hyperslice which adds tracing code to each of a set of methods. Implementing it as a mixin layer is currently not possible. Fortunately, because the idea of refinements as functions is so universal, nothing prevents us theoretically from adding more complicated ways of implementing refinements. There are several directions we will explore in this regard: Instead of composing the subunits in two collectives if they have the same role (composition by role), GenBorg could introduce arbitrary mappings from the roles of one collective to the roles of the other collective. The composition operator $\circ$ could be extended to allow a refinement to modify several units at once, in which case it would probably also play multiple roles. We are also thinking about adding more operators to GenBorg which could, for example, provide other ways of composition or projection (like we've mentioned above).

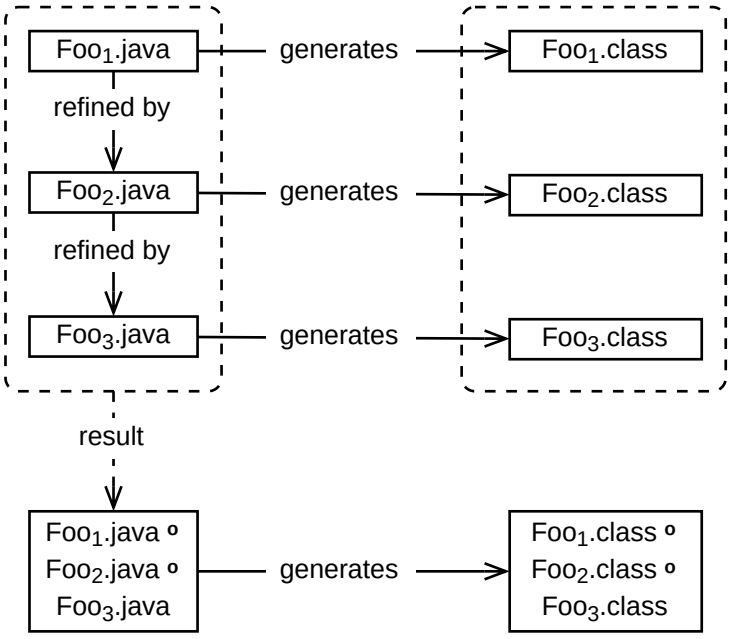


Figure 6.1: The concept of derivation demonstrated by the example of Java source and class files. The class files on the right are derived from source files on the left. Source files can be composed. But to compose class files, one has to compose the source files on which the elements of the composition are based, compose these source files and then *derive* the result class file. Indices have been added to file names (and are not part of them) where different files have the same name.

> A completely different alternative for implementing refinements is to use meta programming [Batory et al., 1993].

Another potential improvement for GenBorg is as well inspired by our work on Probe: A new relation between artifacts. Sometimes one artifact is *derived* from another, which means that the source of the derivation can be composed, but generally not the target. An example of this is a Java class file being derived (i. e., compiled) from source code (figure 6.1). The next version of the GenBorg algebra might explicitly state these relations by integrating Probe and the GenBorg model with the concepts of makefiles.

The type system and the composition operator are other candidates for revision, because the current, inheritance-based, approach to composition can be made more general by dropping the constraint that the result type of a composition $A \circ B$ be a subtype of B. But to ensure correct typing of an equation that contains several compositions, we only have to require that the type of the result of the last, i. e., leftmost, composition be a subtype of the *expected type*. The expected type depends on the context of the equation.

6.2 Probe

Probe is the first implementation of the GenBorg programming paradigm. The main innovations of Probe are how it defines models and how it makes them accessible. First, using directories for defining a model proved to be simple and robust, because it relies on a way of persistence that is natural to current operating systems. Second, we found that navigating a model as a tree only was to constricting, it relied too much on traversing the model's dominant decomposition, collectives, top-down. Representing the model as a relation enabled new ways of browsing that make working with large models more practical. Combining easy definition and navigation of models with pluggable custom implementations of categories and operations makes Probe a probe a powerful build tool. This already delineates the direction current development efforts for Probe are taking: We'd like to further

improve Probe's build functionality by basing it on the *make*-like building tool *Ant* [Foundation, 2001] and by factoring it out as a build kernel. That kernel can be invoked as a command line tool and integrated into automated build processes. Probe's graphical user interface will be just another way of communicating with the kernel.

Other features need to be enhanced, as well: The type system should closer follow GenBorg's lead, traces of the old distinction between constants and functions will be eliminated. Categories as sets of similar units are too closely related to types and will be superceded by that mechanism. Furthermore, model and project directory are so alike in structure that it makes sense to unify both concepts: A model directory instantiates project directories that are again model directories (and can therefore instantiate new project directories etc.). By discarding the constraint that primitive units be files, Probe will finally adopt the GenBorg idea of classes as collectives of methods. An elegant and logical extension of current features, this makes the method level browsable in a fashion that is similar to the SmallTalk development environment.

New capabilities of Probe will borrow from GenVoca or implement current GenBorg constructs: GenVoca's design rules need to take care of semantic constraints regarding composition where types take care of syntactic constraints. Derivation is a relation between units (very often artifacts) that we have mentioned above and that is motivated by make files. Thus, it is only natural to add it to the future Probe kernel which is to become more like a make tool. Many of the tasks a user currently has to perform by hand can be taken care of by Probe. We have experimental tools that automatically type units by parsing the files and directories they are based on. Optimization of equations with regard to certain criteria (such as performance properties), which has already been implemented for GenVoca [Batory et al., 2000a], will be transferred to GenBorg and Probe.

Our vision for the future is to make Probe a tool for programmers and non-programmers alike. Domain experts will get a graphical user interface where they express their needs in domain-specific metaphors. Probe can then write an equation for them that produces the software they want. [Lopez-Herrejon and Batory, 2001] present an example of such a user interface. It uses multiple-choice questions (in a wizard-like fashion) and creates an optimized equation for non-programmers so that they don't have to assemble it by hand (figure 6.2).

The current version of Probe is but a small first step in exploring the possibilities of GenBorg's concepts. Numerous new ideas of how to extend and improve it—some of them mentioned in this chapter—should make Probe an interesting development project for some time to come.

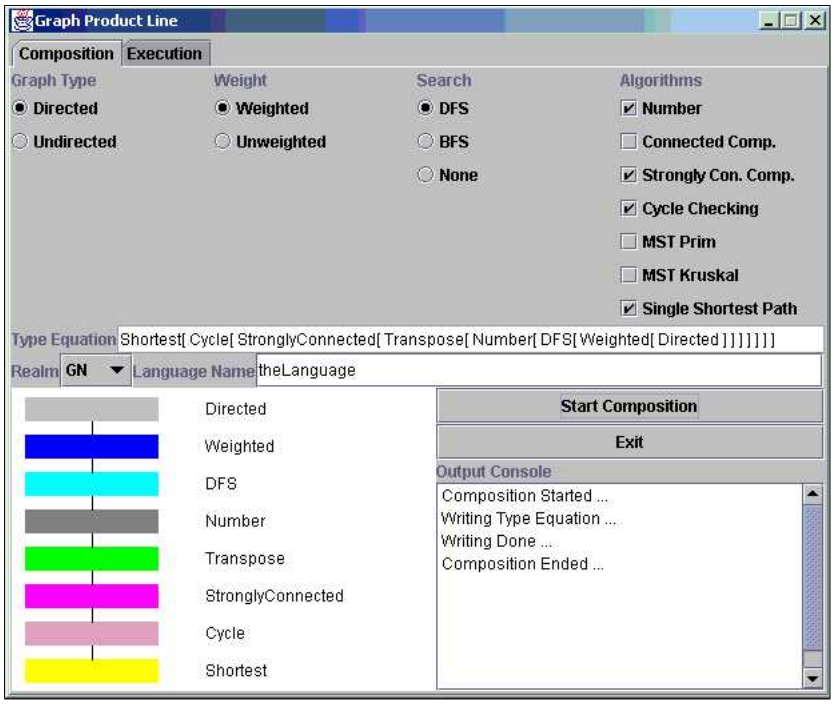


Figure 6.2: An example of a wizard automatically composing an equation for a user. The Wizard bases the equation (in the bottom left corner of the window) on several simple questions it asks the user (top half).

Appendix A

Schema Reference

Section 4.1 contains an introduction to schemas and several examples. Chapter 5 contains a long real-world schema.

A.1 Notation

- *Meta variables* are typeset in an italic roman font. So the value `$`*category*`$` means `$Layer$` if the name of the current category is `Layer`.
- The sections below that describe the tags of a schema or property file contain the entry "Inner tags". This entry shows what tags can be used inside the current tag. It is expressed in *Extended Backus-Naur Format*.

A.2 Rules for Using Variables

There are a few limits on using variables in the value part (the right-hand side) of a property definition if the variable references a property in the same node. There are no limits on referencing properties in ancestor nodes. Note that when matching, there cannot be two consecutive variables without separating literal characters. The reason for this is that every match must be non-ambiguous.

- Only `file` and `role` can introduce new (*match-created*) properties in their value string.
- `file` and `role` have to match-create the same properties.
- The only properties from the same node a value can refer to are the `file`, `role` and match-created properties.

- Only `file` and `role` can introduce new (*match-created*) properties in their value string, because they are the only properties that are ever matched against something. Match-created properties are thus not defined by key and value (their name never appears on the left-hand side of a property definition), but by their name appearing as a variable in a pattern that is matched against a string; that matching gives them their value. Property `*` (or rather, property `WILDCARD`) is usually match-created.
- `file` and `role` have to match-create the same properties. The reason for this is that we have to be able to translate between these two properties. Each of them is alternately used for input and output and, when used for output, has to be *instantiable* (all its variables have to be bound).
- The only properties from the same node a value can refer to are the `file`, `role` and match-created properties. This follows from the fact that when instantiating properties, `file` and

`role` are introduced first. No guarantees are made for the order of introduction of the remaining properties.

A.3 Tag Reference

Below, we enumerate all valid tags for a schema file. The standard name of a schema file is `schema.xml`. It has to be in the root of the model directory.

schema

Attributes: name, dbColumnNames
Inner tags: directory | group | artifact

Attribute	Default	Description
name	–	Gives a human readable name to the schema, ignored by Probe.
dbColumnNames	`"Level 1, ..., Level n-2, Artifact"`	Defines titles of the database columns. n is the height of the category tree.

directory, group, artifact

Attributes: category, role, groupRole, file, class
Inner tags: property* menu (directory | group | artifact)

Attribute	Default	Equivalent property	Description
category	mandatory	–	Assigns this category a name.
file	`*`	`file`	Pattern that specifies what files this category applies to. Cannot be used for groups.
role	`*`	*category*	Role of units created by this category. Cannot be used for groups.
groupRole	mandatory in subcategories of a group	*parent category*	Can only be used by direct subcategories of a group and provides a key for grouping them.
class	Use default implementation	`class`	Qualified class name that provides a custom implementation for units in this category.

The tags `directory`, `group` and `artifact` define categories. A custom implementation for a category of units can be used if specified in the `class` attribute. This class extends either `genborg.base.Collective` (for directories or groups) or `genborg.base.Artifact` (for artifacts), see section D.1 for an example. Its tasks are to implement composition, to indicate how to display the content of a file, etc.

The concept of a *group* needs to be explained. It can be seen as a virtual directory that groups a set of files. Let category `A` be the direct supercategory of a group `B`. The direct subcategories (defined by either directories or files) of `B`, categories $\texttt{C}_1, \ldots, \texttt{C}_n$, define patterns for files in the directory of `A` and a key for each matching file (attribute `groupRole`). Units for files with the same key are put in one collective (an instance of group `B`). Its role is the value of attribute `groupRole`. Properties are stored in a tree that is isomorphic to the unit tree (appendix C), but property nodes of children of a group are the exception to that rule: They are direct children of the parent of a group (and therefore siblings of the group). Only through this trick, we can have the same rules about file patterns for grouped and ungrouped units. Otherwise, children of a group would have been unable to access any ancestor properties in the file pattern (now they can at least read the properties of every non-group ancestor), because it is the file pattern that determines what group a unit belongs to; ancestors would only have

been accessible after matching. Another benefit is that the property tree data structure can use file names for identifying child units, just like property files do, because a path from a node to the root of the tree never contains groups, exactly mirroring the file structure. We also don't have to rearrange property nodes (which are created ahead of their units if there are nested entries in property files) each time we create a group.

property

Attributes: name, value
Inner tags: –

Attribute	Description
name	Key part of the specification of a property default.
value	Value part of the specification of a property default.

Properties given in the schema actually instantiate defaults for properties of units, using strings with embedded variables as explained above.

menu

Attributes: –
Inner tags: operation*

The `menu` tag is a container for operations.

operation

Attributes: menuName, class
Inner tags: –

Attribute	Description
menuName	Gives a human readable name to the operation that can be displayed in menus etc.
class	Specifies the class that implements the operation.

Classes that implement an operation extend the abstract class `genborg.operation.Operation`, see section D.2 for an example.

Appendix B

Property Reference

Chapter 4 contains sections introducing properties (4.2) and property files (4.4) and gives examples for the latter. See section A.1 on the notation used below.

B.1 Properties and Defaults

Property	Default	Allowed in prop file?	Description
category	*	no	The role property.
parent category	*	no	The group role property. Units in a group assign the parent role.
file	*	no	File name of a unit.
type	$*category*$->$*category*$	yes	Type of a unit. Currently not inherited, i. e., if no value is given, a unit always gets the default type, regardless of what has been defined in the ancestor property nodes.
class	*	no	Qualified name of a Java class that provides a customized implementation for this unit.
constant	`false`	yes	Hack that cuts off the domain of a type signature. Unlike `type`, this property is inherited.
ignoreFiles	–	yes	Hack. A space-separated string of file names that should be ignored in this directory and its subdirectories (i. e., it is inherited). Will be integrated into the schema in the future.

B.2 Property File Tags

Below is a list of all valid tags inside a property file. The standard name of a property file is `properties.xml`, it has to be placed inside of the directory whose collective's properties it is to define.

collective

Attributes: –
Inner tags: property* | unit*

`collective` is the root container for everything that's in a property file. Any direct child tag `property` defines properties for the collective that is created from the directory of the property file.

unit

Attributes: file
Inner tags: property* | unit*

Attribute	Description
file	To identify a subunit that is a directory or an artifact, one has to provide the name of its file.
role	To identify a group subunit, one has to provide its role.

The `unit` tag allows one to define properties for (non-directory) artifacts which can't have their own property file (only directories can). Identification is done by file name.

property

Attributes: name, value
Inner tags: –

Attribute	Description
name	The key part of a property definition.
value	The value part of a property definition.

This tag defines a property.

Appendix C

Data Structures

In this appendix, we briefly describe the data structures Probe uses to represent a model in RAM. These are (figure C.1):

- *Category tree*: Saves the schema as a tree of category objects. The category describes common attributes of a set of units, such as how their role names are produced or what operations can be performed on them. Given a file name that matches its pattern, a category object is responsible for creating a unit object.
- *Unit tree*: Hierarchically stores model and equations. A unit object contains references to the category that created it, to the *owner* of the unit, and to its property node. The tree uses owners to signal additions (through composition or when the model is read from disk at the start of the program) to its nodes.
- *Property tree*: Created in parallel with the unit tree. The two trees are isomorphic. Unit objects are created from data files and property objects from property files. Therefore, properties briefly grow ahead of units during creation time if the property file of a collective contains information about nested units. The responsibilities of a property object are to load and save itself if provided with the unit and the unit's owner.
- *Owner*: There are two kinds of owners, a *model owner* and a *project owner*. Thus, an owner tags the source of a unit, i. e., if it was read from a model directory or created through composition. A model owner notifies registered listeners[1] of additions to the unit tree. A project owner additionally logs compositions to the project manifest file. Both owners provide properties and (in the case of the project owner) units with the absolute path name of a directory when they want to save themselves to disk.
- *Database*: Registers as a listener with both owners and stores each new unit in a relation as described in section 4.6.

Figure C.2 shows a UML diagram of the class hierarchy. The abstract superclass `Unit` has been split into parts that become active at different phases in the life of a unit, or sometimes never. `BaseUnit` is always there and contains attributes that every unit has. `ComposableUnit` is only enabled when a unit is the result of a composition. `VisibleUnit` takes care of functionality that is related to public display and saving. *Invisible* units are, for example, intermediate results within an equation: For the equation $f \circ g \circ h \circ a$, there is one unit for every pair of operands. So the unit tree contains units representing $h \circ a$, $g \circ (h \circ a)$ and $f \circ (g \circ (h \circ a))$, but only the last one is visible.

[1] the *observer* part of the *publish-subscribe* pattern, see [Gamma et al., 1995]

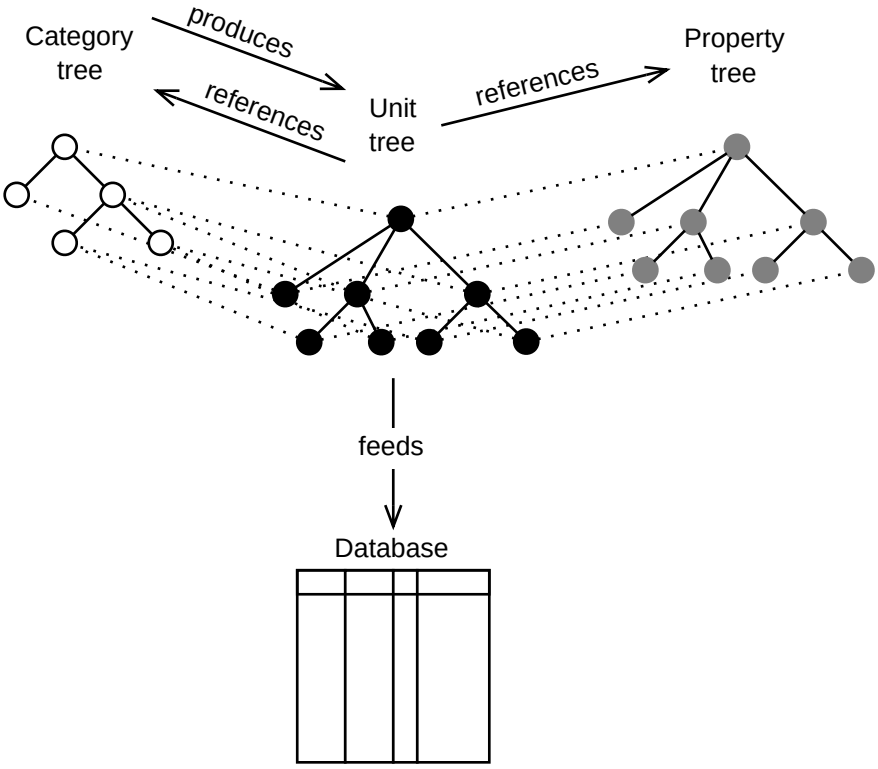


Figure C.1: The data structures Probe uses to represent a model in RAM

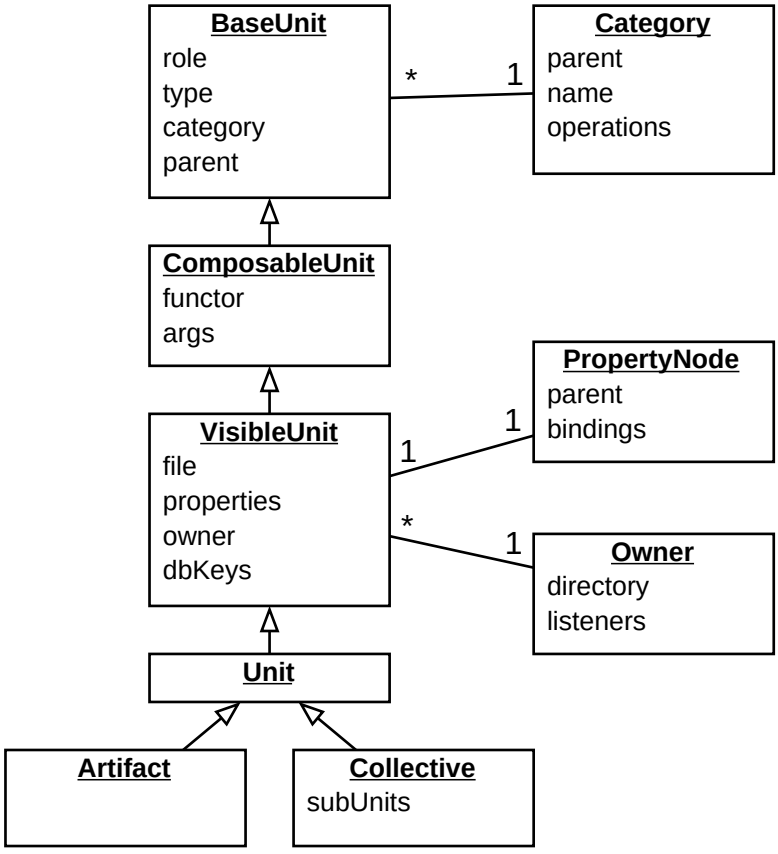


Figure C.2: Class hierarchy of Probe's data structures

Appendix D

Custom Implementations

This chapter gives examples for implementations of categories and operations.

D.1 Example Implementation of a Category

A custom implementation of a category of units overrides either class `genborg.base.Collective` (if the units to be implemented are collectives) or class `genborg.base.Artifact` (if the units are primitive). The following is a custom implementation of an artifact category.

```
public class CustomArtifact extends genborg.base.Artifact {

    /** @overrides Artifact */
    public Unit composeWith(Unit[] args, Collective parentUnit)
        throws IllegalCompositionException
    {
        super(args, parentUnit);
        if ((args.length != 1)
            || (! args[0].getRole().toString().equals("Foo")))
        {
            throw new IllegalCompositionException("Can't compose!");
        }
    }
```

A custom implementation can override any method of its superclasses. In this example, we override `composeWith` so that a `CustomArtifact` can only be composed with one unit whose role is *Foo*. All other compositions are made impossible by throwing an exception. We have to call `super`, because the overridden `composeWith` takes care of many important tasks related to composition.

D.2 Example Implementation of an Operation

Operations only need to override one method of their abstract superclass `Operation`: method `perform( UserInterface, Unit)`. `UserInteface` is an interface that abstracts any kind of interaction with a human or a non-human client (for example, through a graphical or text-based user interface), the `Unit` parameter references the unit this operation has been invoked on. The following operation displays an error whenever it is invoked.

```
public class CustomOperation extends genborg.operation.Operation {
```

```
        public void perform(UserInterface ui, Unit unit) {
            ui.displayError("Operation cannot be performed on unit "
                            + unit.getRole());
        }
    }
```

Bibliography

D. Batory, G. Chen, E. Robertson, and T. Wang. Design wizards and visual programming environments for genvoca generators. *IEEE Transactions on Software Engineering*, pages 441–452, May 2000a.

D. Batory and B. J. Geraci. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering (special issue on software reuse)*, pages 67–82, February 1997.

D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *International Conference on Software Reuse*, pages 117–136, 2000b. URL `ftp://ftp.cs.utexas.edu/pub/predator/fsats.pdf`.

D. Batory, V. Singhal, J. Thomas, and M. Sirkin. Scalable software libraries. In *Proceedings of the ACM SIGSOFT Conference*, December 1993.

T. J. Biggerstaff. A perspective of generative reuse. Technical report, Microsoft Research, December 1997.

K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In *Proceedings of the Dagstuhl-Seminar on Generic Programming*, April 1998.

R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000. URL `http://www0.arc.nasa.gov/~filman/text/oif/aop-is.pdf`.

M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

A. S. Foundation. Ant - an xml-based portable build tool written in pure java, 2001. URL `http://jakarta.apache.org/ant/`.

M. Fowler. *Refactoring.* Addison Wesley, 1999.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, 1995.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison Wesley, 2000.

G. Kiczales. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, October 2001.

G. Kiczales et al. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP).* Springer, 2001.

R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. *Third International Conference on Generative and Component-Based Software Engineering (GCSE)*, 2001.

J. M. Neighbors. *Draco: A Method for Engineering Reusable Software Systems*, chapter 12, pages 295–319. ACM Press Frontier. Addison Wesley, 1989.

H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach, 2000.

V. P. Singhal. *A Programming Language for Writing Domain-Specific Software System Generators.* PhD thesis, Department of Computer Sciences, University of Texas at Austin, September 1996. URL `http://www.cs.utexas.edu/users/schwartz/pub.htm#vivek-thesis`.

Y. Smaragdakis and D. Batory. Implementing layered design with mixin layers. In E. Jul, editor, *Proceedings ECOOP'98*, pages 550–570, Brussels, Belgium, 1998. URL `http://www.cs.utexas.edu/users/schwartz/pub.htm#ecoop-templates`.

P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107–119, May 1999.

D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering*. Addison Wesley, 1999.

D. A. Wheeler. More than a gigabuck: Estimating gnu/linux's size. Web report, June 2001. URL `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`.

N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

M. M. Zloof. Query by example. In *AFIPS NCC*, pages 431–438, 1975.