UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1997-12, NOVEMBER 1997

DESIGN OF A JAVA FRAMEWORK ENGINEERING TOOL

Markku Hakala, Juha Hautamäki, Jyrki Tuomi, Antti Viljamaa, Jukka Viljamaa

University of Tampere
Department of Computer Science
P.O. Box 607
FIN-333101 Tampere, Finland

ISBN 951-44-4256-3
ISSN 0783-6910

Contents

Abstract

Frameworks are reusable designs used to improve the software development process. This report presents a plan to implement the FRED (Framework Editor for Java — Support for Framework Development and Use) tool set for designing and using frameworks. FRED is a joint project between the departments of Computer Science at the University of Helsinki and at the University of Tampere, supported by TEKES and several Finnish industrial partners.

Keywords: Frameworks, Design patterns, CASE Tools

1. Introduction

Traditionally reuse in object-oriented systems has been achieved through the use of *class libraries* [GHJ95, LaN95]. They offer general services, such as basic data structures or stream I/O, but seldom *architectural* or *design level reuse*. That is why the object community has shifted its attention from class libraries to *application frameworks* [CoS95].

An application framework is a set of objects — an application core — that captures the special expertise in some application domain to a reusable form [Deu89, Lew95, Hau97]. A whole working application (or a significant portion of it) can be specialized from this skeleton by adding new features to it. A framework provides the basic functional logic for all applications derived from it, so that the main control of the whole application lies inside the framework. Therefore the framework must be able to call the user defined extension objects as needed. This can be accomplished through the use of inheritance, dynamic binding, object composition, and delegation. Frameworks should finally evolve to *black-box components* the user can easily combine without complete knowledge of their internal structure.

Designing application frameworks is difficult because in reusable systems there are always many important classes that cannot be deduced from the concepts of the application domain. *Design patterns* provide a systematic way to capture the knowledge of application framework experts in an easily accessible way [GHJ95]. Design patterns name, explain, and evaluate important recurring design solutions so that others can reuse them as basic building blocks for new frameworks. This is not enough, however. Without proper tools developers are bound to duplicate their efforts each time they apply a pattern. It may also be difficult for an inexperienced programmer to grasp design patterns and to correctly evaluate their many variant structures. We argue that a *CASE tool* could alleviate these problems by letting the user create, modify, validate, and delete pattern instances. It should also be able to trace and backtrack design decisions to make it easy to try out different solutions for a given design problem.

Construction is not the only difficult phase encountered when utilizing frameworks. Deriving applications from a framework is a complicated process, too. Normally a framework specializer must study and understand the framework thoroughly before she can use it. We believe that tool support also for creating framework documentation and for following specialization instructions could be valuable, because framework specialization would thus become a guided process where mechanical coding could be done automatically. One framework might have several different specialization instructions, e.g., for advanced and novice users.

In this report we specify a CASE tool (*FRED — Framework Editor for Java*) that helps in constructing and using application frameworks written in the Java programming language [Sun97a]. First we describe the tool's general architecture (chapter 2). In chapter 3 we study the user interfaces of its four major components *Pattern Composer*, *Framework Composer*, *Application Composer*, and *Visualization*. A brief tutorial is provided in chapter 4 to illustrate potential ways to employ these tools. After that we look at the system from the implementor's point of view (chapter 5) and consider possibilities to use FRED as a part of a modern software development environment (chapter 6). Finally we discuss related work and draw some conclusions in chapter 7. Few examples of formalizing design patterns are given in appendix A. Appendix B describes the tag language used in code templates[1]. An alphabetical listing of all FRED's classes is given in appendix C.

This report is a product of the *FRED (Framework Editor for Java — Support for Framework Development and Use)* project. It is a joint project between the departments of Computer Science at the University of Tampere and at the University of Helsinki, supported by TEKES and several Finnish industrial partners: Dycom, ICL Data, Major Blue Company, Nokia Telecommunications, Nokia Research Center, Novo Group, Stonesoft, Sun Microsystems, Telecom Finland, Tiedonhallinta, TT-Tieto, and Valmet Automation.

[1] While having the same name, templates of FRED are completely different from templates of C++. A code template in FRED is a skeleton for class definition which will be filled in with a special editor (*template editor)*.

2. The Overall Architecture of the FRED Toolset

A CASE tool for application framework development and usage should assist the designer, support experimentation, eliminate routine work, and guide framework specialization [ACL97, BFV96]. In this report we propose a framework engineering toolset (*FRED* — Framework Editor For Java) that meets these goals and provides tight integration of design and implementation phases of software development. Figure 2.1 represents the overall architecture of the FRED toolset. In the figure we see how a working application consists of one or more frameworks and some additional application code. A part of the framework code is automatically generated from the employed design patterns. Also a portion of the application may be provided automatically during the specialization phase. The missing code fragments have to be written by the framework user or imported from an external source.

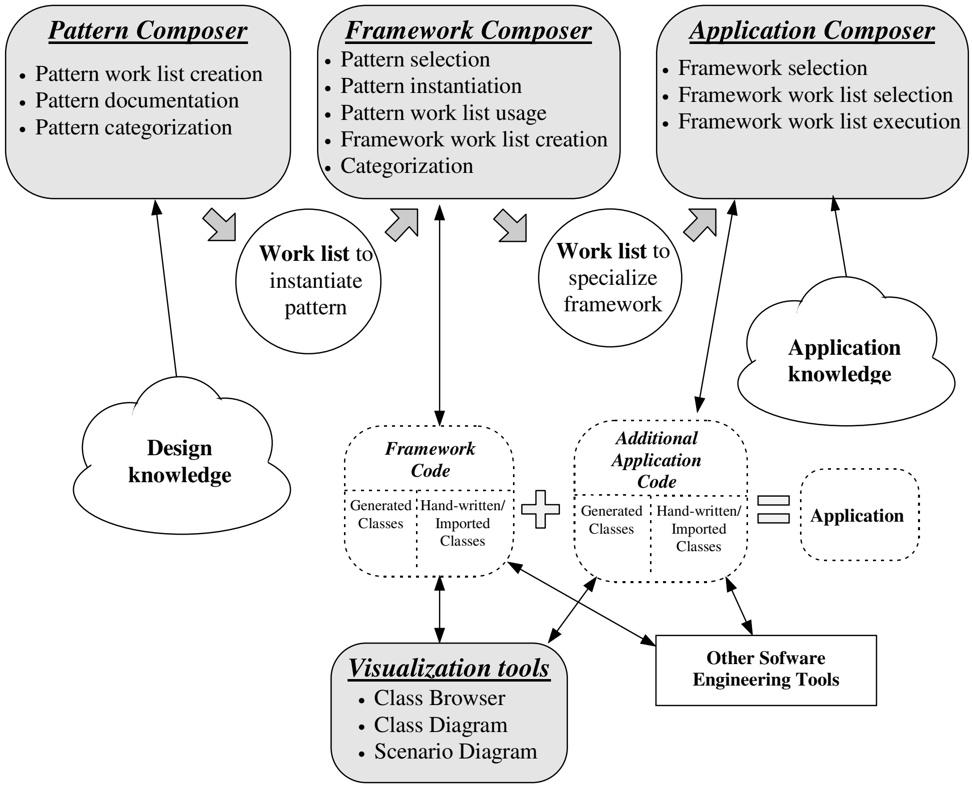

Figure 2.1: The overall architecture of the FRED toolset

The FRED toolset has four major components (marked with gray background in figure 2.1). *Pattern Composer* converts existing design knowledge into pattern descriptions with a uniform structure. It integrates many of the ideas found in the existing pattern-based CASE tools (see, e.g., [Vil97c]). The roles of a pattern are represented as *work lists*. The pattern's work list includes *templates* for the classes and methods that implement it. They must be detailed enough to enable automatic code generation. The work lists store also *constraint information* that is used to check the implementation's conformance to the pattern structure. Various *documents*

— for example, a set of interlinked HTML pages — can be associated with the patterns and they are also *categorized* according to different criteria for supporting intuitive browsing and easy selection.

Framework Composer aims at making framework construction an easier task for software developers. It is based on the idea of systematically *instantiating* and applying design patterns. The tool provides powerful control over design and implementation decisions so that the framework developer can try out different alternatives. The developer can select design patterns to use in her framework simply by browsing the design pattern descriptions and documentation or by using a *dedicated wizard* that helps the selection.

Once the framework developer has found the patterns that meet her needs she must instantiate them and bind implementation-level names to their roles. *Framework Composer* then creates *pattern instances* for keeping track of these bindings. The new design pattern instance and its role bindings must of course be compatible with existing instances and associations. The pattern instances remain in the system to document the structure of the framework as it evolves. The pattern instances and their work lists can also be used to generate the implementation code if it does not already exist.

In *Framework Composer* the structure of a framework is described in terms of the design patterns it includes. In the framework specialization phase this is not enough, because the design patterns do not explicitly describe the purpose of the framework and how it should be used. Thus, the framework developer must provide work lists and documentation for the intended framework users. The created pattern instances — especially their work lists — can be used as a base for these *framework work lists*. In addition, some higher-level work items guiding the whole specialization process can be specified. Work lists can also be processed and saved for later use. The work list mechanism offers a flexible way for the framework developer to provide detailed and dedicated framework specialization instructions for a variety of users.

Application Composer includes *Framework Browser* which is used to select one of the available frameworks and one of its work list alternatives as a base for the application to be developed. After selecting the appropriate work list the framework specializer starts executing it. Here the selected work list is used as a kind of instruction sequence; by executing tasks step-by-step the application developer specializes the current framework. The work list itself is constructed by the framework developer using *Framework Composer*. This is because only the framework developer knows the necessary steps needed to obtain an application from the framework. During the execution dedicated *tools* assist the user to make the required specializations and object compositions. The code is edited with a special Java editor that is built on top of an editor framework being designed at the University of Tampere [Hak97]. It will include special facilities for code transformations, *hot spot* editing, and change propagation across related code fragments. The framework developer may have also implemented special tools for some tasks. For example, a graphical editor for drawing state machines might be supplied for a networking framework user's convenience. The graphical representations of parts of the system and their source code counterparts are kept consistent.

The framework documentation must support users with varying backgrounds and different levels of experience in using the framework [MCK97]. Methods to support program understanding include documentation as well as program understanding and reverse engineering tools such as analyzers and visualization tools. *Application Composer* must satisfy the requirements of the following user groups identified by Johnson [Joh94]:

1) Users deciding which framework to use.
2) Users wanting to build a typical application. They want to learn how to use the framework in order to develop a standard application. The amount of required reading has to be minimized.
3) Users wanting to go beyond the typical use. They need the detailed descriptions of the framework design and the implementation algorithms. They need to be able to find very specific information quickly.

Each step in the framework specialization process should have documentation associated with it, and this information must be easily available. Of course, the framework developer must provide this information first. The first and the second user group need to have short descriptions and examples which guide them quickly through the specialization process. For those who want to go beyond the typical use there must be links to more specific documentation. Also tools extracting knowledge directly from the system must be provided.

The quality of the available framework work lists and the documentation is up to the framework developer. However, design patterns can help in their making: each design pattern involves the actions needed to make it functional and each pattern is formalized as a sequence of work items. These items build up the back bone of work lists provided for a given framework.

Finally, the *Visualization* subsystem provides representations of both the static structure and dynamic behavior of an application or part of it. *Class Browser* represents framework and application classes as a tree hierarchy. It also shows the properties (methods, data attributes, role bindings, etc.) of the selected class. *Class Diagram* represents an application as a static class chart and *Scenario Diagram* offers a dynamic visualization of the execution of the application. The design patterns are used to *highlight* only the information that is currently of interest, and to filter out all unnecessary details.

Other Software Engineering Tools which could be interfaced with FRED would typically be tools like specialized source code editors, Java compilers, debuggers, etc. Currently, FRED has not any specific interface for attaching external tools; these could possibly need extra functionality and supporting features withn FRED. On the other hand, some tools — like a Java compiler — would probably need very little specific support from FRED.

3. The User Interface

In this chapter we take a closer look at FRED's user interface. We describe its functionality mainly by showing how different commands can be invoked using menubars and buttons. Note that also popup menus, keyboard short cuts, and drag and drop features will be implemented to provide easier access to the features. All screen shots presented in this document were created with JBuilder's [Bor97] graphical editor just to demonstrate the user interface components. Currently these components have no functionality and their appearance may change.

FRED can be run as a regular Java application, but we are also planning to enable running FRED as a Java applet. Some of FREDs individual components can also be used as stand alone applications. Figure 3.1 represents the toolbar from which all major FRED components can be launched. The main user interface components are *Pattern Composer*, *Framework Composer*, and *Application Composer*. *Pattern Composer* is used for creating formalized design patterns and their categories. With *Framework Composer* the user applies design patterns to build frameworks, and constructs work lists for assisting application developers utilizing the framework. *Application Composer* is used for deriving working applications from frameworks by executing selected work lists. In addition to these tools, FRED has components that provide visualizations of the system's structure and behavior.

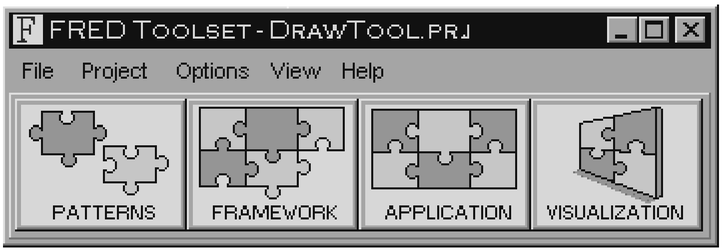

Figure 3.1: The toolbar for the FRED toolset

The menubar of FRED's main toolbar includes menus for handling projects, their files, general options, and for opening and closing other FRED windows. In the *File* menu one can open the *New Project* dialog to specify a name and directories for a new project (see figure 3.2). The user can name several directories to define all possible locations for source and class files. She can also insert all files into the project at the same time. The user can open an existing project with the *Join to Project* dialog (see figure 3.2). The possibility to join to an existing project is especially useful when several developers are working on the same project. In that case the project's files are kept in a shared directory, possibly on a remote server, and individual developers fetch working copies of the files when required. This implies a need for a version control and check in/ check out functions (see also chapters 6.2 and 6.3). The remaining selections in the *File* menu implement the usual functionality to save the project (possibly under a different name) and to exit the whole FRED toolset.

In the *Project* menu the user can add classes to the project using the default file selector dialog. If a directory is selected, all classes in the directory and its subdirectories are added. Files are always inserted into packages according to their location in the file hierarchy. No files are ever moved or renamed when they are added to the project. With the *Project* menu's *View Project Files* command the user can launch *Class Browser* in order to graphically manage the files of the project[2].

The *Options* menu includes, for example, options for saving the layout of the FRED windows. The *View* menu includes standard selections for showing or hiding individual or all windows, and the *Help* menu provides hypertext help and product information.

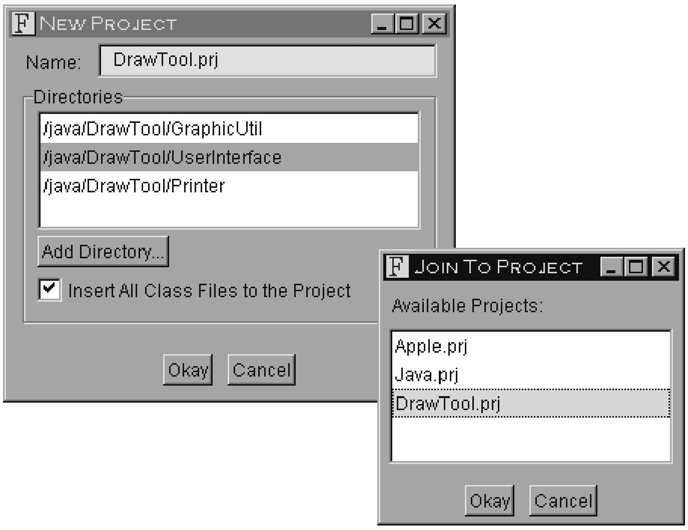

Figure 3.2: Dialogs of the toolbar

3.1 Pattern Composer

In the FRED toolset a design pattern is formalized as a three-part data structure. Each pattern has a set of *work lists* (one for each pattern variant), *category information*, and *documentation*. A work list describes the participants of a pattern and how the pattern should be used. When instantiating the pattern each participant is bound to some implementation-level entity, such as a class or a method.

Pattern Composer (see figure 3.3) is used for creating and modifying design patterns, assigning patterns to certain categories, and managing the categories. *Pattern Composer* lists the available design patterns and their categories, as well as the pattern variations (different work lists) and documents of the selected pattern each in their own section.

2 *Class Browser* is opened with its *File Structure* hierarchy selected (see figure 3.17).

The *Categories* section displays the pattern categories as a tree structure. After every category name (leaf node) there is a float value between 0 and 1 denoting how well the selected pattern suits the category. When adding new categories the pattern developer[3] must provide a textual description about the category. Also a qualifying value concerning the new category must be associated with every pattern[4].

All four lists in *Pattern Composer* can be modified with the *New*, *Change*, and *Delete* buttons. When applied to *Work List Variants* the *New* and *Change* buttons open *Work List Editor* (described later in this chapter, see figure 3.5). *Work List Variants* are useful in situations where the pattern designer wishes to create different work lists, e.g., for providing different default behaviors for a pattern.

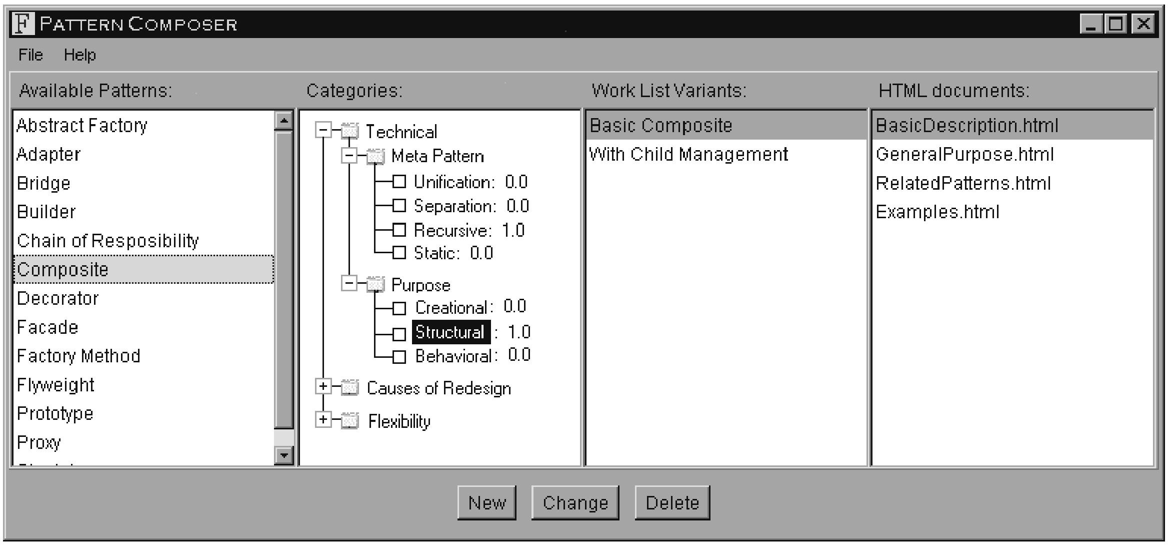

Figure 3.3: *Pattern Composer*

The hypertext documentation for a design pattern consists of an arbitrary set of interlinked HTML pages. Since we want to be able to reuse possibly existing documentation, we cannot set any specific restrictions on the structure of a pattern document, but we expect that the documents more or less follow the pattern description structure suggested in [GHJ95]. Figure 3.4 shows an example of what an HTML-based design pattern document could look like.

3 We call those users of the FRED toolset who define new design patterns *pattern developers* to differentiate them from *pattern users* who apply existing patterns in their design work.

4 The user can specify every new value separately or use a neutral default value.

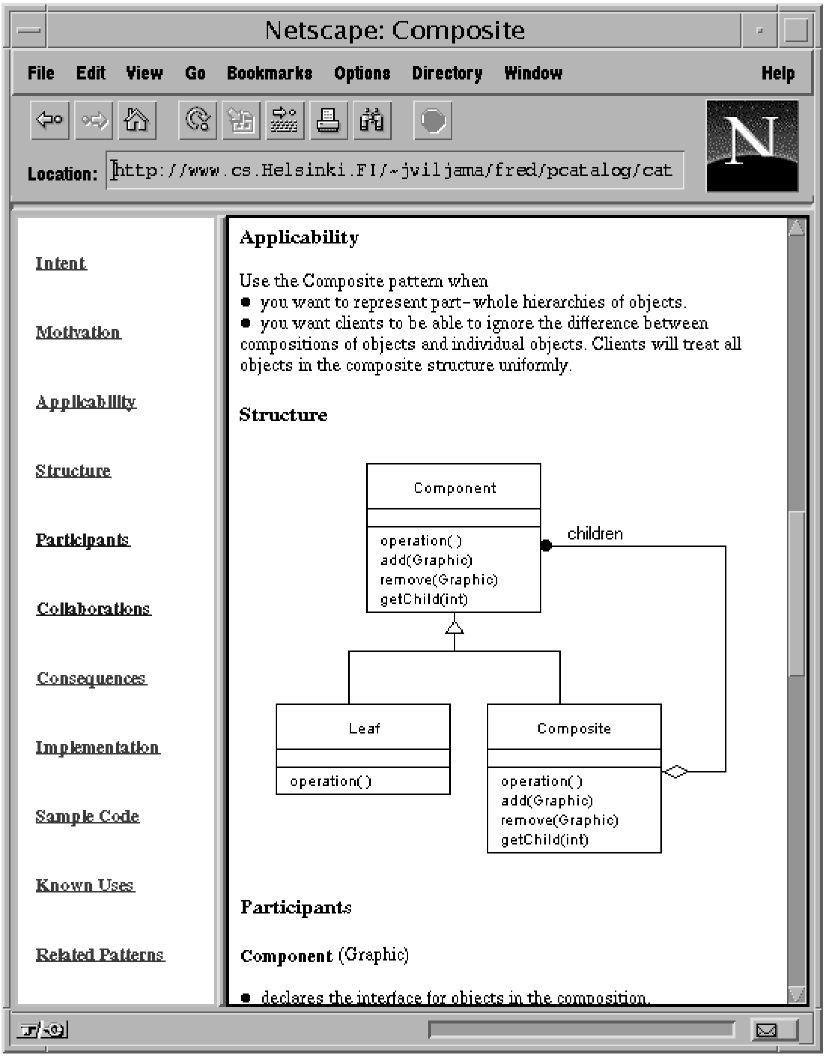

Figure 3.4: An example of design pattern documentation

Work List Editor

A work list consists of work items. Each work item describes a small well-documented and automatically aided task that has to be executed in order to proceed in framework specialization (or design pattern instantiation) process. Work items include *code templates* that describe the Java code generated from them and the restrictions for further code modifications. They also define *formal constraints* that must be checked when validating the use of a pattern. In the framework specialization phase work lists guide the application developer so that she applies the framework as intended. When the user has implemented the task described by a work item, the item is said to be *done* (or executed). Items in a work list can be done in any order unless a specific execution order is requested. A work list item typically has a name, description, optional restrictions, example, and the tool — with its initialization parameters — responsible for the item execution.

Each work item will have some kind of editor tool associated with it. By default all items are associated with *Template Editor* (see chapter 3.3), but more specialized, perhaps visual, tools can also be provided. Framework user completes the Java code generated from a work item by using its (special) editor tool. The tool sees that the user obeys the editing restrictions of the item.

Work List Editor (see figure 3.5) creates and modifies work lists. It is needed for composing new design patterns and for building work lists describing the specialization instructions of a framework. The user can save work lists into files and also import work items from existing files to create new work lists. Work items have different kinds of associations between them: they may be subitems of each other and they may have dependencies related to their execution order[5]. These associations are visualized and manipulated in the tree view on the left (figure 3.5).[6] The items are always shown in the currently selected hierarchical order. The selected hierarchy can be manipulated using drag-and-drop. All modifications to the items and their properties are done using the *Add*, *Change*, and *Delete* buttons or the *Edit* menu.

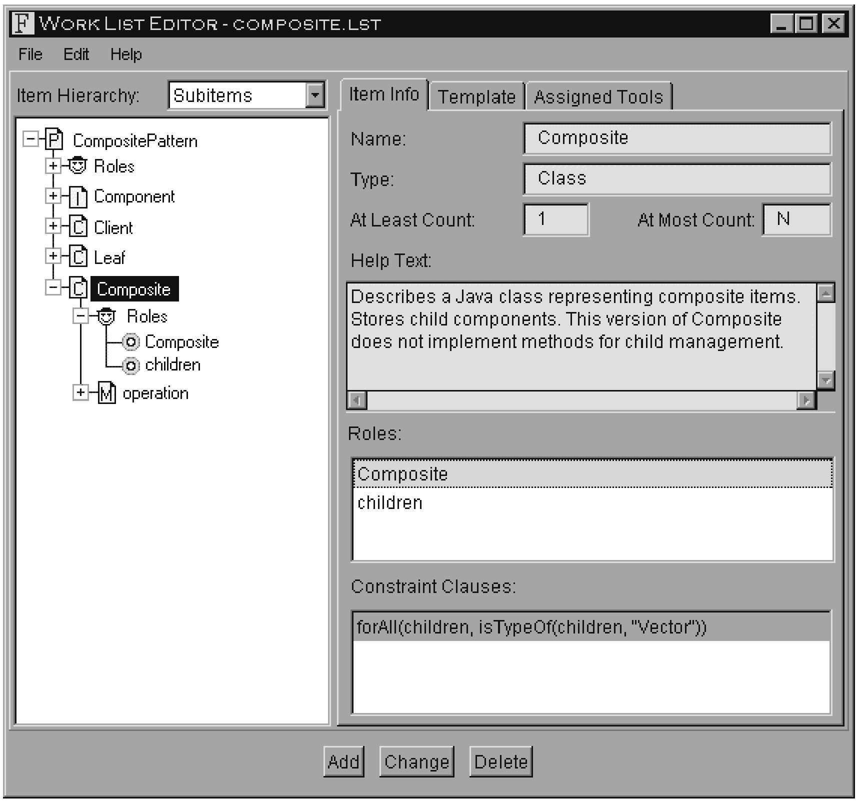

Figure 3.5: *Work List Editor*

---

5 For instance, items can be *connected* with each other. This means that the subitem templates of the item that is connected to some other item are expanded into the template of the item that owns the connection relation. This mechanism is useful, for example, when a class plays roles in several design pattern instances, and we want to include the methods described in subtemplates inside the same class template.

6 In figure 3.5 only the work items are visible. However, the user can also view the *work item instances* (if any exist). This is necessary especially when constructing a work list for a framework, where some parts of the work list are already executed. See chapter 3.2 for more information.

Existing items can be used as a base for new items (they just must be saved under a different name). *Work List Editor* supports also conversions between different types of items. For instance, a class item can be converted into an interface item and vice versa. This mechanism updates the template files of the items accordingly. When creating an item the user can also decide to start with an item initialized to, e.g., a subclass or a method of a certain class, or to a class implementing a certain interface.

All properties of the selected work item are visible in the tab folders on the left. The *Item Info* tab shows the basic properties of the item — name, type, cardinality[7], help text, the roles defined in the item, and constraint clauses[8] for checking that the implementation code does not conflict with the item. The *Template* tab includes information about the code template associated with the item. The user can launch *Template Maker* (described later in this chapter) from the tab to create or modify the template. The *Assigned Tools* tab shows the editor tools available for editing the item during its execution.

Template Maker

The code template of a work item is divided into *fields* surrounded by *tags*. Tags describe the properties of a field, for example, whether or not the user can edit the field, a default value for the field, and a help text to be displayed as a hint for the user on how to edit the field.

Most of the templates are actually textual Java interfaces, classes, methods, or data attributes equipped with some additional control information. Thus, it is natural that *Template Maker* (used for creating templates) is an intelligent text editor capable of inserting certain tag marks into source code. It can be compared to a visual HTML editor. *Template Maker* is invoked from the *Template* tab of *Work List Editor*.

When the user creates a new template for a work item, she can take a piece of existing Java code (a class definition, for instance) to act as a skeleton for the new template or she can start from scratch. To complete the template, the user must specify role definitions and references[9], default values for certain elements, and editing restrictions on some text areas.

Figure 3.6 represents *Template Maker* as the user is editing one of the Composite pattern's templates. The text window shows the template as the user will see it, i.e. without the tags. The tags are revealed by the underlinings in the text. For example, role definitions are marked with solid and role references with dashed lines. The user opens a *Tag Properties* dialog to create and modify tags associated with the selected text area.

---

7 Cardinality indicates how many times a work item can be executed, i.e. how many work item instances can be created from it.

8 Constraints are given using a simple textual language (see chapter 5.2.5).

9 Roles denote those parts in the templates that must be consistent in the implementation. For example, when the user gives a name for a common interface for some implementor classes, a role is defined. That same role is then referenced in the templates of the implementor classes to ensure that they declare the correct interface in their `implements` clause.

Besides role definitions and references, a tag may concern, for example, expansion of other template files into the tag area and editing restrictions (see appendix B for more information).

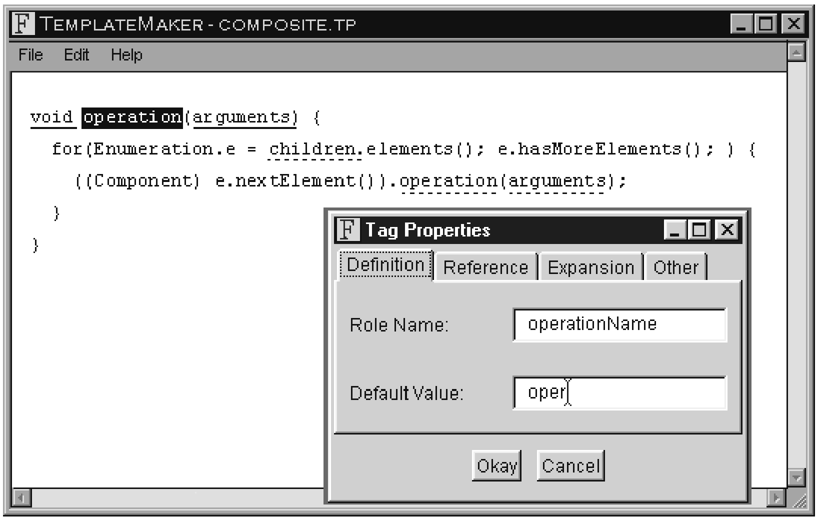

Figure 3.6: *Template Maker*

3.2 Framework Composer

Framework Composer (see figure 3.7) is used to develop a framework by instantiating design patterns, associating pattern roles with implementation code (code can also be generated), and creating work lists for the application developer. It can also be used just to visualize current associations between the patterns and the implementation. With the three buttons — *New*, *Change*, and *Delete* — the user can create and modify pattern instances, roles and their bindings, as well as work lists for the framework under construction. The same functions can be applied from the *Edit* menu.

The *Pattern Instances* section on the left shows the names of the design pattern instances as a tree structure. Pattern instances can be organized in different ways (see various categories in *Pattern Selection Wizard*, figure 3.9). The chosen classification forms the first level in the tree structure (the root node always stands for the open project). The second level of nodes is formed by design patterns that have instances. The leaf level consists of the actual design pattern instances. If the symbol in front of the instance name is a filled donut (in contrast to an unfilled donut), it indicates that all the roles of the instance are properly assigned to some implementor classes. When the user wants to add a new pattern instance, *Pattern Selection Wizard* is opened and the user can choose which kind of pattern (and variation) to create.

The *Items & Roles* section in the middle represents class and member roles of the chosen design pattern instance as a tree structure. The tree consists of three kinds of nodes: *work item instances* (illustrated with a small page symbol and a letter describing the type of the work item; *P* for pattern, *C* for class, *I* for interface, and *M* for method), *role lists* (marked with a face symbol), and *actual roles* (shown with a donut symbol). At the end of an actual role name there is the name of the implementation level entity that the role is bound to. In a design pattern there may be several implementors for the same role. For example, the (first) role of the work item *Leaf* in the *Composite* pattern in figure 3.7 is implemented by classes *Line* and *Rectangle*. At the end of a work item instance name there is the index number for that instance and the binding of the first role of the item's role list[10].

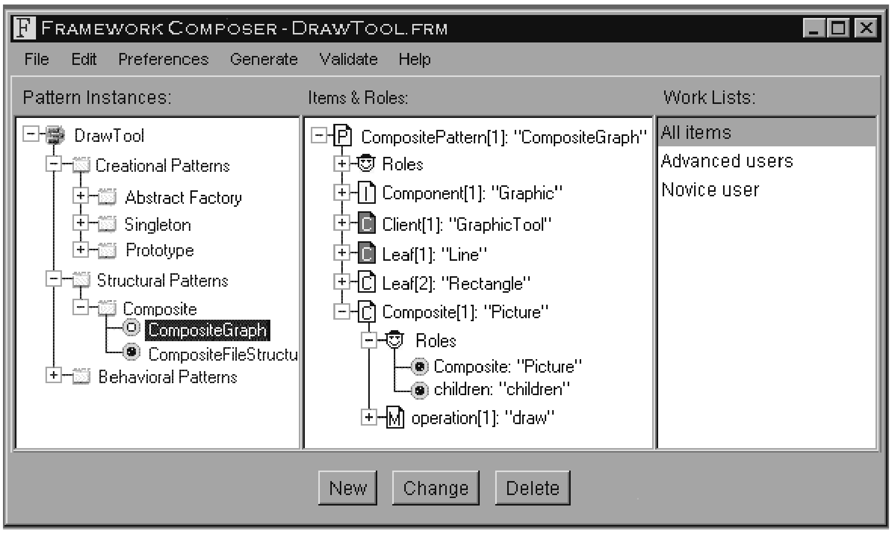

Figure 3.7: *Framework Composer*

When an implementation entity (e.g., a class) is associated with a role in a design pattern instance, some constraints must be obeyed. This typically involves checking the associated classes, members, and inheritance relationships. The exact constraint rules are of course unique for each design pattern and they are given in the work list of the pattern. If a role binding does not satisfy the given constraints, the user is informed. After all roles of a work item and its subitems have been bound, it can be validated. A ready and validated item is marked with a *dark* page symbol. Figure 3.8 shows a dialog representing the properties of a selected work item instance. In the dialog the user can see the validation status of the item and she can also specify some filtering options to simplify the tree representation.

The *WorkList* section represents all work lists intended for the users of the current framework. It always includes the *All Items* work list that combines the work items of the instantiated design patterns. The framework developer can construct the final work lists for the application developers by reusing the work items from *All Items* and by adding suitable new items. In figure 3.7, the framework developer has created two work lists and named them "Advanced users" and "Novice user". It is upto the framework developer to decide what items these work lists contain. Here, e.g., the "Advanced users" work list might enable the application developer to control some advanced features of the resulting application where defaults typically are sufficient.

[10] For example, work item instance *Composite[1]* has the string "*Picture*" at the end of its name, because its first role (*Composite*) is bound to that entity.

The *New* and *Change* buttons are used to launch *Work List Editor* for creating and modifying work lists. Each of these work lists is a copy of the original *All Items* work list in the sense that it includes all the items and their instances together with all role bindings. The framework developer can, however, hide the unnecessary details like those items whose maximal cardinality has been reached and most of the item instances. The framework developer can, for instance, leave just a guiding representative work item instance that the user can then clone as long as the cardinalities are not violated. In the user interface this is reflected by the two modes of the item hierarchy tree view in *Work List Editor*. Figure 3.5 represented only the work items without their instances, but work items can also be shown together with their instances. This mode is particularly useful when developing a work list for a framework.

Framework Composer has *File, Edit*, *Preferences*, *Generate*, *Validate*, and *Help* menus. *File* menu is used for opening and saving work lists for the framework under construction[11]. The *Help* menu provides hypertext assistance in the tool's usage. In the *Preferences* menu the user can open dialogs to edit general options or options related to this window. General options might include, for example, options for compiling code with debugging information or compiling running demos of design patterns. The view preferences on the other hand are specified through a dialog similar to the *View Preferences* dialog of *Class Browser* (see figure 3.16). The *Generate* menu includes a command for generating code for all classes present in the design pattern instances and selected from the *Items & Roles* list. The *Validate* menu is similar to the *Generate* menu. It includes commands to validate selected items.

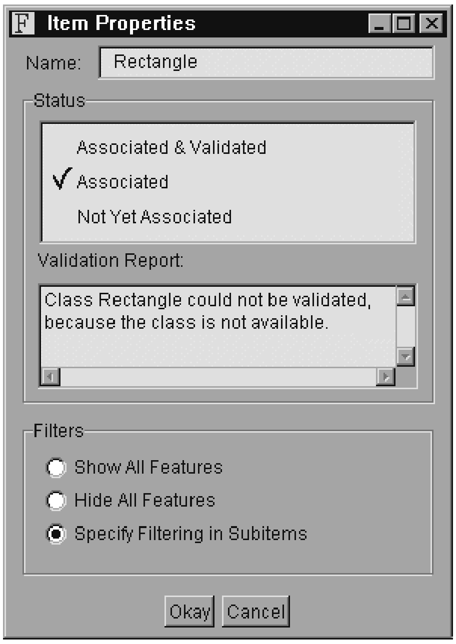

Figure 3.8: Dialog for displaying the properties of a work item

11 Note that this also saves the work item instances and their role bindings. So, a work list may be already executed to some extent.

Pattern Selection Wizard

Pattern Selection Wizard (see figure 3.9) is used for instantiating design patterns. It also assists in finding a suitable pattern for the desired purpose. The user can select any of the patterns in the *Available Patterns* section and view its basic intent. The tool also lists the pattern's ratings in the pattern categories. With the *Instantiate* button the user can instantiate the selected pattern and with the *Show HTML Description* button she can view its HTML documentation. If the user is not familiar with design patterns or does not immediately know which pattern to instantiate, she can open the *Pattern Filter* section with the *Search Help* button.

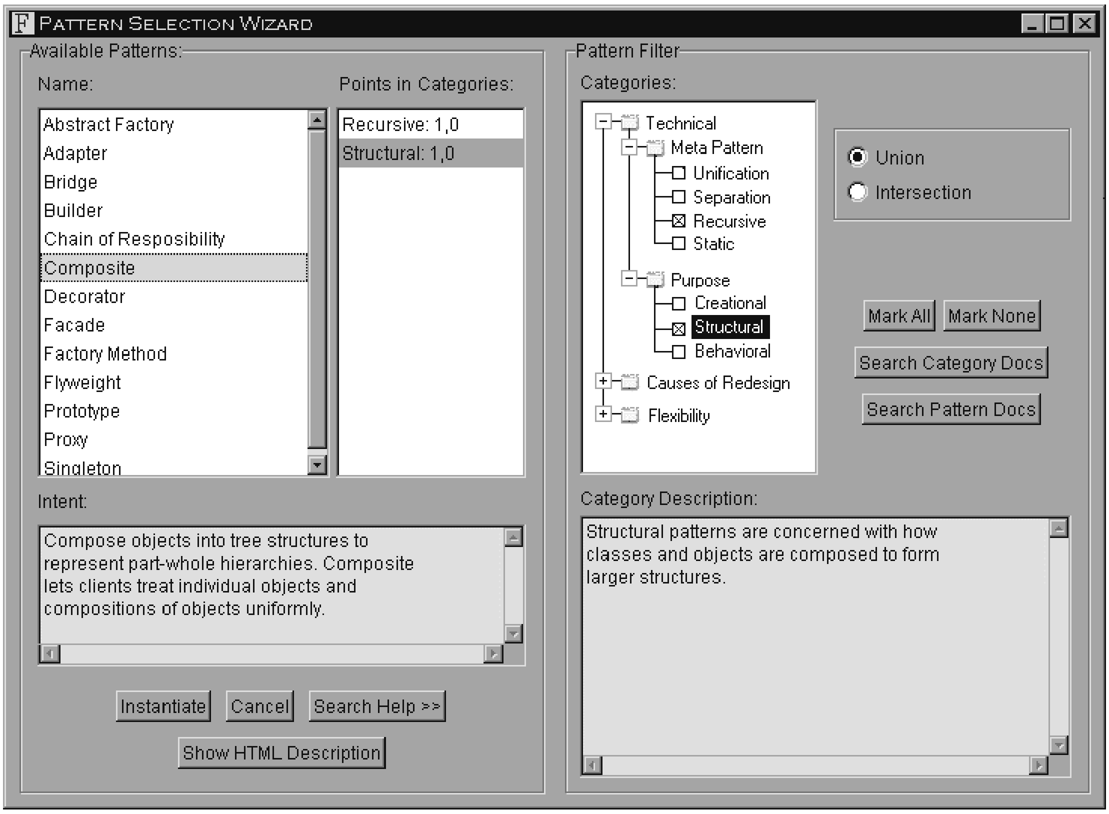

Figure 3.9: *Pattern Selection Wizard*

The *Pattern Filter* section lists all available categories (and subcategories) as a tree structure. It allows the framework developer to mark the categories from which patterns are included into the *Available Patterns* section[12]. The result of the filtering consists of the matching patterns sorted according to their applicability as expressed by their category values. With the radio buttons the developer can choose whether to display a union or an intersection of the patterns matching different categories. *Category Description* shows a short help text concerning the currently selected category. The user can also make text searches (using the *Search Pattern Docs* and

[12] Note that a pattern can belong to several categories. If a category is not marked, it does not necessarily mean that no patterns matching that category are included in the resulting list — a pattern might still be included because it belongs to some other category that has been selected.

Search Category Docs buttons) to find an individual pattern or a pattern category that might help in her design problem. The two text sources available for searching are the HTML documents associated with each pattern and the descriptions of the design pattern categories. Text search facilities will utilize the existing Web search engines, such as AltaVista.

In figure 3.9, there are three main pattern categories: *Technical*, *Causes of Redesign*, and *Flexibility*. The *Technical* category includes subcategories *Meta Pattern* and *Purpose*. The *Purpose* category allows the framework developer to select all *Creational*, *Structural*, and *Behavioral* patterns as described in [GHJ95]. The *Meta Pattern* category allows the selection of *Unification*, *Separation*, and *Recursive Composition* patterns, as defined by Pree in [Pre95]. We have included an additional category — *Static* — for those patterns whose principal idea does not rely on dynamic binding (e.g., *Facade*). The *Causes of Redesign* category allows to choose solution patterns for commonly encountered redesign requirements [GHJ95]. The *Flexibility* category divides patterns into groups according to whether modifying the behavior of an application using a specific pattern requires an application restart or not.

3.3 Application Composer

While *Framework Composer* uses design patterns as framework building blocks, *Application Composer*, in turn, assists in applying frameworks in order to construct working applications. *Application Composer* helps an application developer to identify *hot spots* in frameworks. Hot spots are the places where the framework will be extended (see, e.g., [Pre95]), i.e., where the application developer must add his code. For example, hot spots can be:

- New subclasses that must be derived from the existing ones. For instance, there could be a superclass for animals and the user could create new animals by deriving subclasses.
- Methods that can be overridden. This includes the methods that *must be* overridden and the methods that *may be* overridden.
- Attributes that must be initialized
- Objects that must be instantiated.
- Application initialization..

In the next subchapter we first cover the general use of *Application Composer*. The user interface views are discussed in subchapters *Framework Browser*, *Application Wizard*, *Class Factory*, *Template Editor*, and *Special Editors*.

Using Application Composer

The user interface of *Application Composer* contains four main views: *Framework Browser*, *Application Wizard*, *Class Factory* and *Template Editor*. In addition, there can be a number of *Special Editors* to supply higher-level assistance on applying certain frameworks. For example, a networking framework could be equipped with a graphical editor to specify finite state machines defining the behavior of protocol entities. Another, more general example (illustrated in figure 3.15) is *Composition Tool* used for object composition. These tools are optional, i.e., the user can refine a hot spot directly in *Template Editor*, too. The control flow among the components of *Application Composer* is shown in figure 3.10

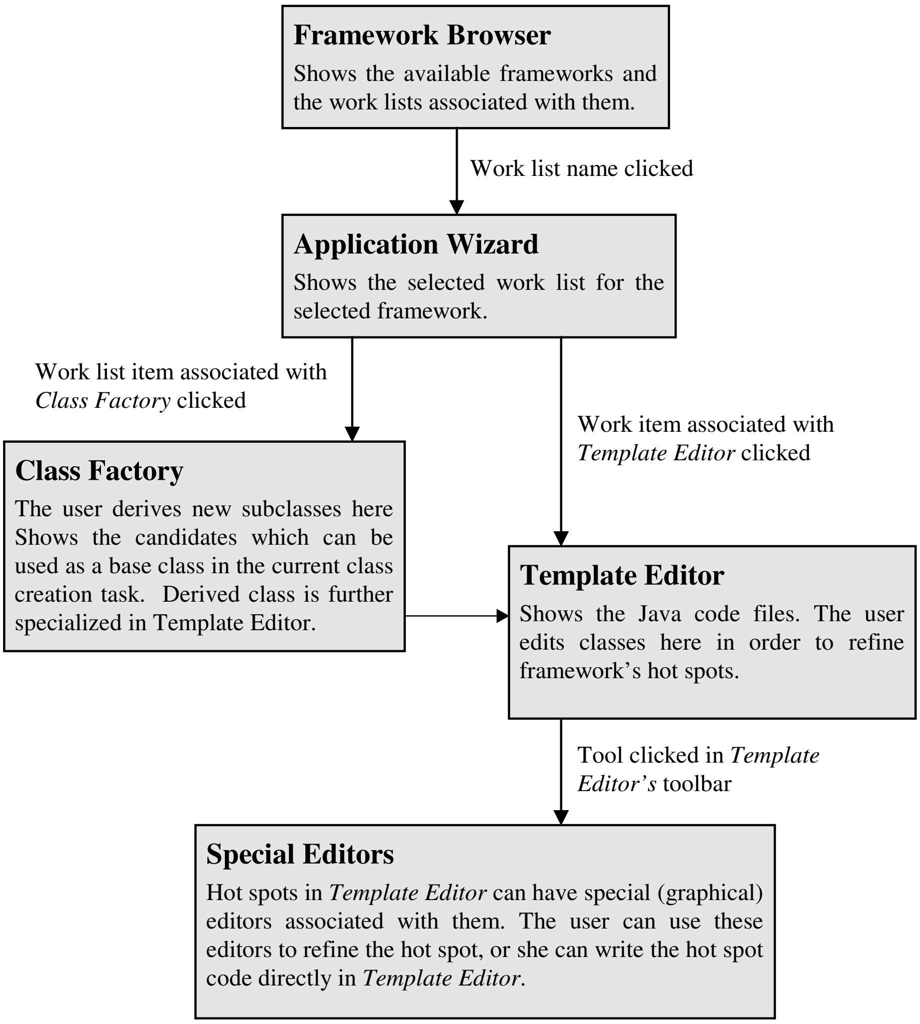

Figure 3.10: The control flow among the components of *Application Composer*

Application Composer is typically used as follows: first an appropriate framework with a suitable work list is selected in *Framework Browser*. Then the application developer executes the corresponding work list step by step in *Application Wizard*. A single work list action can open, for example, *Template Editor* or *Class Factory*. *Template Editor* allows the user to view the classes and to modify them where applicable, i.e. specialized classes. *Class Factory*, in turn, is intended to create new classes which are usually derived from the framework's component classes.

Class Factory is activated as a result of executing a work item which declares that the application developer must derive a new subclass from one or more base class candidates in the framework component hierarchy. It provides the user with a restricted set of possible base classes and lets the user simultaneously view source code implementations of multiple classes with *Template Editor*. Also, the framework designer has provided helpful documentation which will aid the application developer when she is creating the subclass. In order to create subclasses, each base class candidate is associated with a template file providing a skeleton implementation for derived classes. A new application class with the skeleton implementation can then be modified by the user in *Template Editor*.

Framework Browser

When an application is going to be constructed, the application developer can look for a suitable framework with *Framework Browser* shown in figure 3.11. It is a view which contains a list of available frameworks and their work lists. Each framework and work list has a short description which helps the application developer to make her choice. Note that a framework can have multiple work lists which are shown under the corresponding framework name in the *Frameworks* section. There can be, e.g., a tutorial work list for learning the framework and a work list for a very specific type of applications. When the user chooses a framework by selecting one of its work lists the *Application Wizard* is opened for that framework.

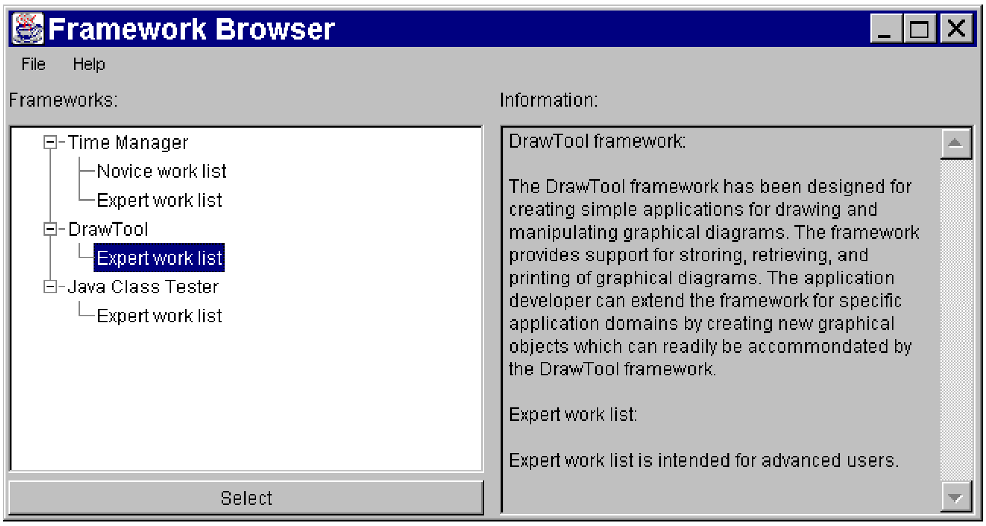

Figure 3.11: *Framework Browser*

Framework Browser contains the following items:

- The *Frameworks* section contains a list of available frameworks and work lists associated with them.
- The *Information* section contains text which helps the application developer to quickly assess the scope of the activated framework. It can also have text associated with the currently selected work list. With this information the application developer should know which is the best framework and the best work list for her purposes.
- The *Select* button opens *Application Wizard* for the currently selected framework and its work list (the selection is actually made by choosing a work list).
- The *File* menu has commands to open framework collections and unfinished framework specialization projects.
- The *Help* menu provides access to the documentation files if the user wants to know more about various frameworks and work lists.

In figure 3.11, we have selected the *Draw Tool* framework and its *Expert work list* to be used as an application base (note that in reality the work list names can be much more illustrative). We will use this framework as an example case in the following chapters. Typical steps in specializing the *Draw Tool* framework with *Application Composer* are the following:

- Derive a new subclass for the drawing window frame.
- Optionally create new graphical items which can be drawn and manipulated by the application user.
- Associate the new graphical items with new menu and toolbar items in the drawing window. That is, if you have created a class *Circle* you must also add a new button for it to the drawing application's toolbar.

Application Wizard

Application Wizard, see figure 3.12, is opened when a particular framework work list has been selected in *Framework Browser*. The idea is to construct an application by performing actions in the work list step by step. Each work list action launches either *Class Factory*, *Template Editor*, or some other special tool.

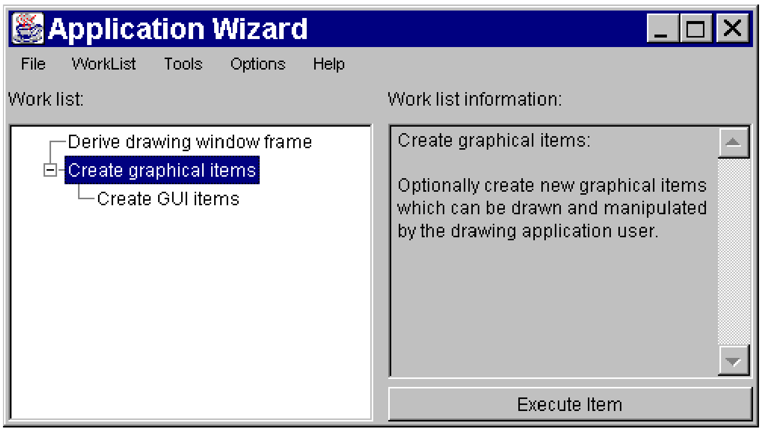

Figure 3.12: *Application Wizard*

The contents of *Application Wizard* are as follows:

- The *Work list* tree view contains the current work list which is used to specialize the framework.
- The *Work list information* section contains information associated with the currently selected work item. It helps the application developer to start the specialization process. This information is especially directed to the users wanting to build a typical application quickly and it can also be used to browse the work items without actually executing them.
- The *Execute Item* button opens *Class Factory* or *Template Editor* depending on the selected work item.
- The *File* menu has commands for saving, opening, and closing projects and for creating the final application with the Java compiler.
- The *WorkList* menu provides commands to control the work list.
- The *Tools* menu provides various tools, like *Class Browser*.
- The *Options* menu makes it possible to change project settings, etc.
- The *Help* menu provides more framework and work list documentation and instructions.

In figure 3.12, the application developer has selected the work item *Create graphical items*. Pressing the *Execute Item* button causes *Class Factory* dialog — which has been associated with this work item — to be opened. Here the developer can derive new subclasses and optionally refine their hot spots using *Template Editor*. Note that the *Create graphical items* work item has a dependent work item called *Create GUI items*. The GUI items (toolbar buttons, menu items) enable the final application user to create new graphic items which are of the derived subclass. If the framework developer takes the time to create suitable templates, the task of writing the Java code to create these GUI items could be done completely automatically by *Application Composer*.

The work list guides the application developer to perform all necessary actions and also ensures that the actions are done in proper order. Executing a work item can result in new work items being dynamically generated, thus tasks may be added dynamically to the work list and they can also be removed by *Application Composer*. Typically, new actions are added when the user executes existing ones, e.g., when the user creates a new subclass, this subclass often needs some extra actions, typically requiring the user to fill in the hot spots. Note that the work list modifications are automated; the user can concentrate her attention on the specialization process.

As we have seen, the work list contains all the necessary steps needed to construct an application from the framework. Some of these actions can be executed directly, without the user. These are called *auto-activated actions* and they are typically some predefined initializations and code generations performed when the framework is selected. These kinds of tasks are removed from the work list after they have been executed. Other actions are executed by the application developer and they are called *user-activated actions*. They can include tasks like creating and editing individual classes.

Class Factory

It is a common situation that there is an abstract superclass (or an interface) and the application developer must derive new subclasses from it (or create a class that implements the interface). In *Application Composer* this is handled with *Class Factory* shown in figure 3.13. The view contains possible superclass candidates, a class description, and an information section which tells the user what she is expected to do. A new subclass is created by selecting one of the displayed superclass candidates and pushing the *New* button. New work items may be added to *Application Wizard*'s work list to refine the created classes.

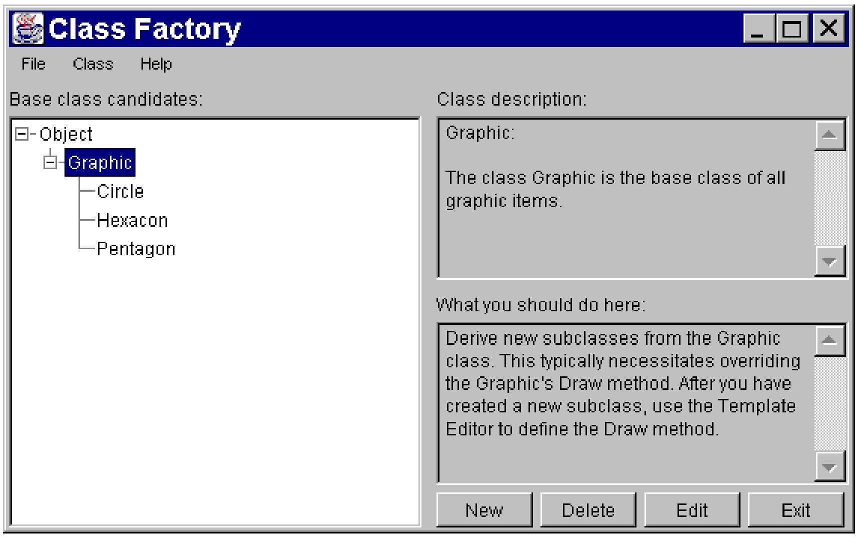

Figure 3.13: *Class Factory*

Class Factory includes the following components:

- The *Base class candidates* section contains superclass candidates and classes that have already been created. Java interfaces will be displayed at the top-level hierarchy level with the classes implementing a particular interface displayed below the interface. An appropriate class (eligible for derivation) needs to be selected when the application developer creates a new class. Consider the example class hierarchy in figure 3.13 where the classes *Circle*, *Hexagon* and *Pentagon* have been derived and defined by the user. The class *Graphic* is a framework class having a template associated with it. By using the template, *Application Composer* knows how to build subclasses.

- The *Class description* section contains a description about the currently selected class.
- The *What you should do here* section contains documentation and examples.
- The *New* button creates a new subclass. A name and a short description are asked for the class. The superclass will be the currently selected class in the *Base class candidates* section. Each superclass has a template which is used when *Application Composer* builds a default body for the new class.
- The *Delete* button removes a user-created class from the *Base class candidates* section. The original framework classes cannot be removed.
- The *Edit* button opens *Template Editor* for the selected class. With the editor the application developer can inspect and modify the code, if allowed.
- The *Exit* button closes the view.
- The *File* menu has commands to save the project and to close the view.
- The *Class* menu provides commands to create, edit, and delete classes in the *Base class candidates* section.
- The *Help* menu provides more documentation and instructions.

To demonstrate the use of *Class Factory* view, we have created three subclasses derived from the *Graphic* base class in figure 3.13. Depending on the user's preferences, *Template Editor* may be opened automatically for hot spot refinement each time she derives a new subclass. Note that *Application Composer* will check that all hot spots have been refined when the user requests to compile the final application with the Java compiler.

Template Editor

Template Editor used in the FRED toolset uses a text editor framework being developed within the project [Hak97]. *Template Editor* makes full use of the features provided by the text editor framework. In the following we will briefly describe some of its capabilities. The tag language used with *Template Editor* is presented in appendix B.

Syntax highlighting of lexical elements of the Java language, such as keywords, identifiers, operators, numbers, and strings, is implemented by identifying them as *tokens* that know how to paint themselves. The appearance, including font face, style, size, and color, of any group of tokens is fully customizable.

Template Editor can format various parts of the document differently. Consider, for example, the convention to include HTML formatting tags inside Java formal comments. Because of this convention, it is useful to display multi-line comments according to typesetting rules determined by the HTML language. This is possible because different *formatters* are responsible for formatting the comments and the actual source code. Nothing between the comment marks is split into tokens according to the Java lexical structure. Instead, the formatter splits the comments into tokens according to HTML language; any HTML tags are identified as tokens and the text between the tags is separated into words and white space. Comments are therefore displayed according to the HTML typesetting rules.

Also a set of interactive *triggers* is implemented. When the user moves the caret inside a comment, for instance, different kinds of actions are possible compared to editing the actual source code. This is due to the event handling mechanism that queries the possible commands whenever a new formatting *scope* defining new options is entered by the caret. One visible result of such queries is that whenever a comment scope is entered, toolbar is decorated with buttons that allow rich text formatting and links. There are buttons for most basic HTML tags, such as bold text and font size and color changes. When a range of characters is selected from the comment region, these buttons insert appropriate tags at the beginning and end of the selected range. The range is reformatted and changes are reflected on the view resulting as typesetting changes. Also links can be inserted in a similar way. A link to any package, class or method in the Java classpath or to an external URL can be inserted. If the link is later clicked with the mouse, the appropriate Java source code file is loaded and displayed by the editor.

Template Editor maintains a logical structure of the document in the form of classes and methods that are contained in the edited document. This substructure is a partial parse tree of the current compilation unit. This structure can be used to provide a small but convenient tree view controller that can be used to jump into specific location in the text.

In addition to this basic feature, elements of this structure can be folded just like described in [MöK95]. Each element of this logical structure on screen is preceded by a folder image. By clicking this image, the user may fold the body of the class, method or interface leaving only the signature visible.

A special tag can be inserted into text to enable or disable the editing of selected regions. This tag is never actually visible on the screen, but the formatter uses it to insert a scope disabling all edit actions over the next brace, bracket, or parenthesis-delimited section. This feature makes it possible for *Template Editor* to guide the user and restrict her editing only to those code fragments where appropriate.

There is also a mechanism to insert arbitrary elements — a Java applet representing a class diagram of the subsystem, for instance — into the source code. Other features of *Template Editor* include, traditional editing with clipboard functions and drag and drop, use of triggers to implement the role binding mechanism of code templates, and window management of multiple files and views.

The flexible parts of a framework developed with the FRED toolset are modified in *Template Editor*. By adding her own code the application developer specializes individual classes and assembles the whole application. A sketch of *Template Editor* is shown in figure 3.14. The class *Circle* in the figure has one hot spot method (`Draw`) which is currently being edited.

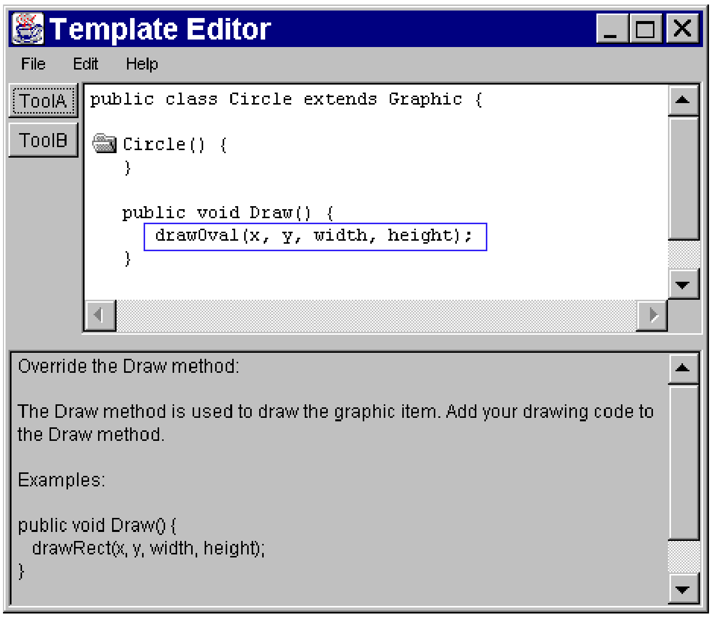

Figure 3.14: *Template Editor*

The contents of *Template Editor* are as follows:

- *Text area* shows the source code of the class to be modified. The hot spots are marked and one can refine a particular hot spot simply by writing the code into it. Each hot spot is associated with various tools (*Tool palette*) and documentation (*Information area*)[13].
- *Tool palette* contains buttons for various *Special Editors* associated with the currently selected hot spot. Tools are optional (one can write the hot spot code directly in *Template Editor*) but they can offer a more comfortable interface to the selected hot spot. Available tools can vary depending on the selected hot spot.
- *Information area* offers guidance to the application developer. It tells what to do with the selected hot spot. It contains documentation and examples and it can also contain links to technical reports, related documents, etc.
- The *File* menu has commands to save the project and to close the view.

[13] These facilities are provided by the framework developer.

- The *Edit* menu has commands to cut, copy, and paste the text.
- The *Help* menu provides access to the framework documentation.

Note that *Template Editor* can be activated from any view which displays a class hierarchy, e.g., *Class Browser* and *Class Factory*. This makes it easy for the developer to view and edit classes at any point of the development process. *Template Editor* could also provide some basic code transformations and refactorings, such as changing an interface definition to a class definition and vice versa.

Java compilation and error correction can be done within the same *Application Composer* session. FRED will provide a means where compilation errors are reported to the user and locating the erroneous code is done automatically for the user. Note that in general it is not possible to always check that individual methods actually do what they are supposed to do, e.g., if the user does not follow the instructions for application specialization given in the work list.

Special Editor Tools

Application Composer allows the user to edit classes either at the source code level or with high-level graphical tools. The term *Special Editor Tools* refers to various tools associated with the framework at hand. Each framework can have special tools for making the refinement process easier. The user is not forced to use these tools — she can always edit the selected hot spot directly in *Template Editor*.

One of the *Special Editor Tools* is *Composition Tool*. It offers a graphical user interface for object composition, where all component alternatives are shown. The object composition is done by dragging the available components to the component slots. Another way is to write the composition code directly in *Template Editor*. However, *Composition Tool* automates code writing and offers a more comfortable way for composing applications from *black-box* frameworks. Figure 3.15 illustrates the features of *Composition Tool* in general level.

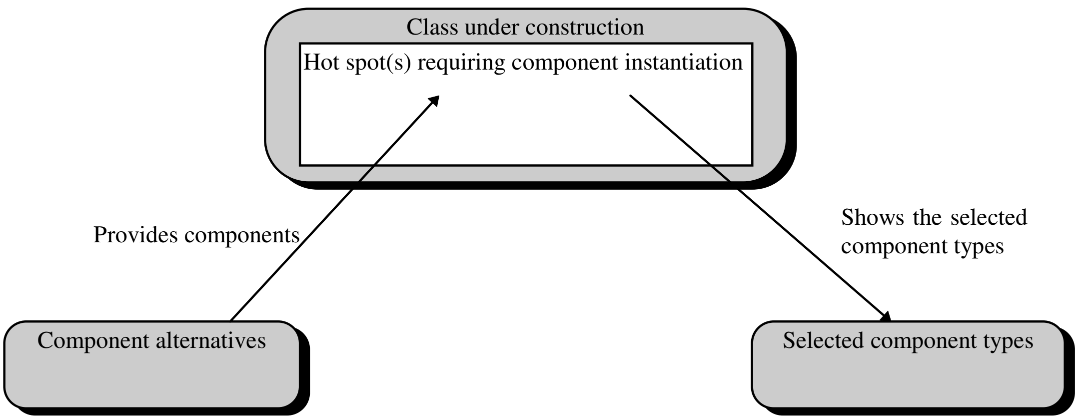

Figure 3.15: The features of *Composition Tool*

Composition Tool can be used to aid in the following tasks: (1) creating declarations and instances of component classes, (2) specifying and modifying constructor call parameters, and (3) creating and modifying method calls and specifying their parameters.

Some of the considerations associated with the use of various *Special Editor Tools* are:

- The framework developer decides in advance which special editor — if any — will be associated with a particular hot spot.
- Each *Special Editor Tool* returns a portion of Java code which is then placed to the current hot spot by *Application Composer*.
- Each *Special Editor Tool* must know how to initialize itself. For example, if the user has modified by hand a particular hot spot where the Java code has been generated by a tool, re-entering the tool for that hot spot may result in a situation where the editor tool is unable to recognize the code as generated by itself. This may or may not cause problems for the tool as the method for initializing the special editor tool has been provided by the framework author.

3.4 Visualization

FRED's visualization tools illustrate the structure and behavior of the system under construction. *Class Browser* represents a static view of the implementation elements, *Class Diagram* provides the same information in more abstract chart format, and *Scenario Diagram* animates the method calls between system's objects. The views in the visual tools may be abstracted according to the used design patterns. Note that *Scenario Diagrams* can only be used for visualizing complete, appropriately instrumented applications, while *Class Browser* and *Class Diagram* views can be used even with incomplete applications.

Class Browser

Class Browser shows the implementation-level entities sorted according to a selected hierarchy. It has two separate section (see figure 3.16). The section on the left allows to browse Java code based either on the *package and class file structure* (as currently chosen in the figure), the *class inheritance structure*, or the *method calls of classes*. In each case the browser shows a tree representation of the chosen hierarchy. When examining the packages and class files the root node of the tree stands for the open project and its children represent different packages and source code directories[14]. The leaves are Java classes or interfaces. When looking at the class inheritance hierarchy, the root node is always the *Object* class, which is the ultimate superclass of every Java

[14] The user can provide a directory where to look for the class if it is not part of the current project. The user can also at any time choose to view files in an arbitrary directory. Class and source files are by default in the same directories, but the user can also provide a different root directory for the source files.

class. Nodes and leaves are either user-defined classes or classes belonging to some Java package. In the call hierarchy all nodes represent classes and their methods.

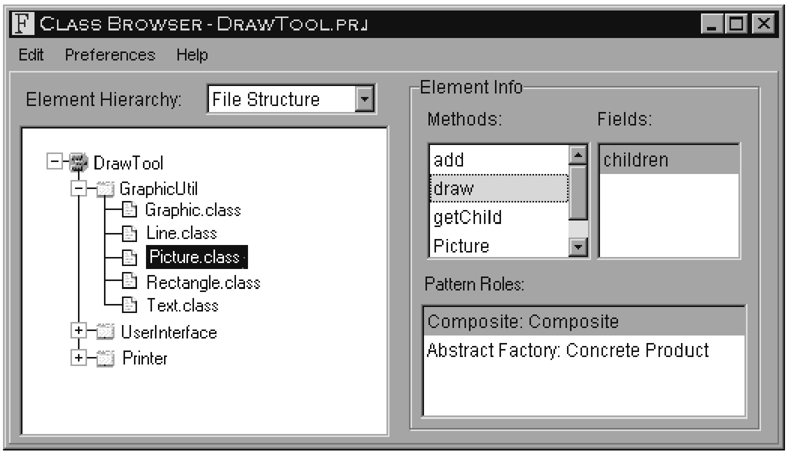

Figure 3.16: *Class Browser*

When the user selects a class from the tree view, lists of its members (methods and fields) and roles in design pattern instances are displayed on the right. If the user selects a role in the *Pattern Roles* section, the corresponding design pattern instance and role are shown in *Framework Composer*. When the user selects any of the items in the *Methods* or *Fields* section, the corresponding source code line is shown in *Template Editor*.

Class Browser has a menubar with *Edit*, *Preferences*, and *Help* menus. From the *Edit* menu the user can open dialogs to specify properties of the project, selected package, selected class, and selected member, respectively. In each of these dialogs the user can specify the so called filtering options (see, e.g., the *Edit Project* dialog in figure 3.17). They affect the visualization of the system in *Class Diagram* and *Scenario Diagram* windows. The options are for showing or hiding the item or for deferring the decision to individual subitems of the selected item. In the *Edit Project* dialog the user can also change the name of the project. The last command in the *Edit* menu associates the selected item (either a class or a member of the selected class) with the current role in *Framework Composer*.

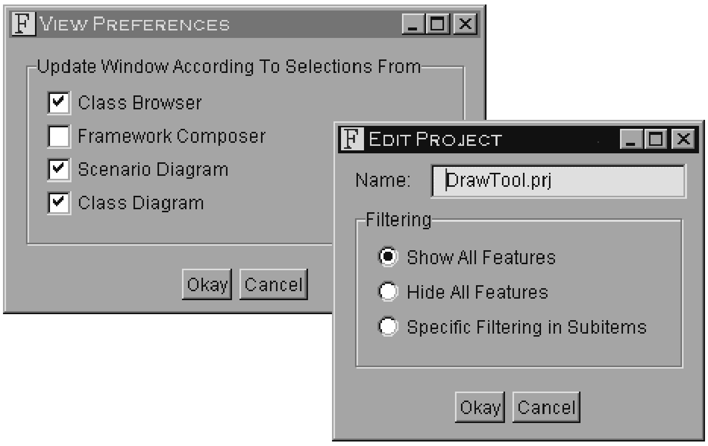

Figure 3.17: Dialogs of *Class Browser*

From the *Preferences* menu the user can pop up dialogs to adjust general and view-specific options. For example, in the *View Preferences* dialog (see figure 3.17) the user can choose how *Class Browser* is updated when selections in other windows are changed. In this way *Class Browser* can be synchronized to *Framework Composer* and to *Scenario Diagram* and *Class Diagram* windows.

Class Diagram

The *Class Diagram* window shows an OMT (or perhaps a UML) class diagram of the current project (see figure 3.18). The user must decide which design pattern instances are shown in the diagram. The user can add individual classes to the diagram and modify the layout by dragging classes to desired locations. Default layouts are supplied for design patterns.

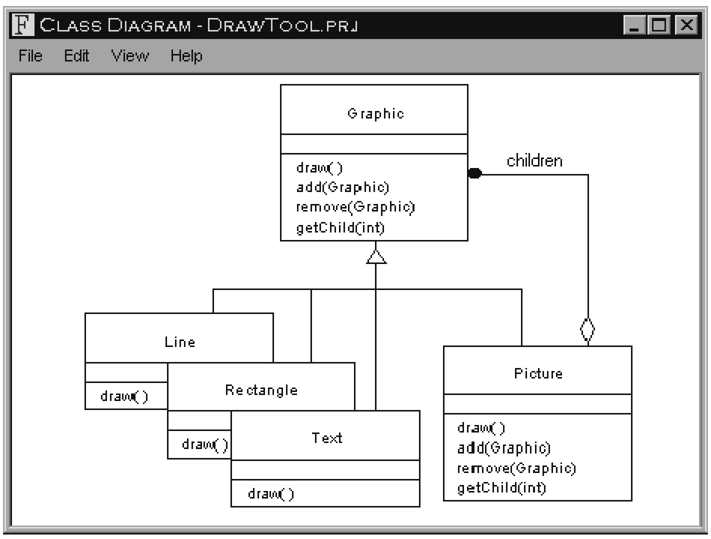

Figure 3.18: The *Class Diagram* window

The *Facade* design pattern [GHJ95] plays a special role in filtering design information, because it can be used to hide all the details of the subsystem that lies behind it. Visually this means that the whole subsystem can be collapsed into the facade class so that only it remains visible. If the user wishes she can of course open up the subsystem to study its internal details. Also subclass hierarchies can be collapsed so that only the base class is visible representing all its specializations also. If there exist associations that are specific to only some subclass, they are lifted up to the superclass and marked with a restriction indicating the appropriate subclass's name. The design pattern instances, classes, and class members in corresponding windows (*Class Browser* and *Framework Composer*) can be used to select class- and pattern-based default values for visualization.

Class Diagram has *File*, *Edit*, *View*, and *Help* menus. From the *File* menu the user can open, save, and print diagrams using the usual default dialogs. When *Class Diagram* window is used from within FRED, it always represents the current project and it is opened, saved, and updated automatically along with the whole project. If the user chooses to open a new diagram or save the diagram under a new name, a separate window is created for the new diagram. *Class Diagram* can also be run as a stand alone program to view saved diagrams.

In the *Edit* menu the user can insert new classes to the diagram or remove classes from it. New classes must be part of the project and removed classes are deleted only from the diagram representation — not from the project itself. It is not possible to create new classes through the *Class Diagram* window. Note that editing of existing classes is also limited to their visual appearance in the diagram. It is not possible, for example, to edit members or associations of an existing class. The *Edit* menu contains also selections to edit properties of the project, selected class, or selected member. These menu items bring up similar dialogs as the corresponding commands in *Class Browser*'s *Edit* menu.

The *View* menu contains selections to zoom the diagram and also to adjust the size of the diagram so that the whole diagram is visible on the screen. The *View | Filters...* command pops up a dialog where the user can define which selections she wishes to use when filtering out unnecessary details from the diagram. These options include at least the filter settings of *Class Browser* or those from *Framework Composer*. The *View | Update...* selection on the other hand opens a dialog in which the user can specify which other windows this window is synchronized to. Finally the *Help* menu provides hypertext assistance on *Class Diagram* usage.

Scenario Diagram

The *Scenario Diagram* window (figure 3.19) displays animation of the dynamic behavior of a running system created with the FRED tools. The window shows (some of) the system's objects as boxes in the upper part of the window. Time runs downwards in the diagram and operation calls made from one object to another are represented as horizontal arrows. The scenario view visualizes also the life time of the objects by showing their creation with a special kind of "lightning arrow". Deletion of objects is not shown, because Java uses garbage collection instead of explicit object destruction.

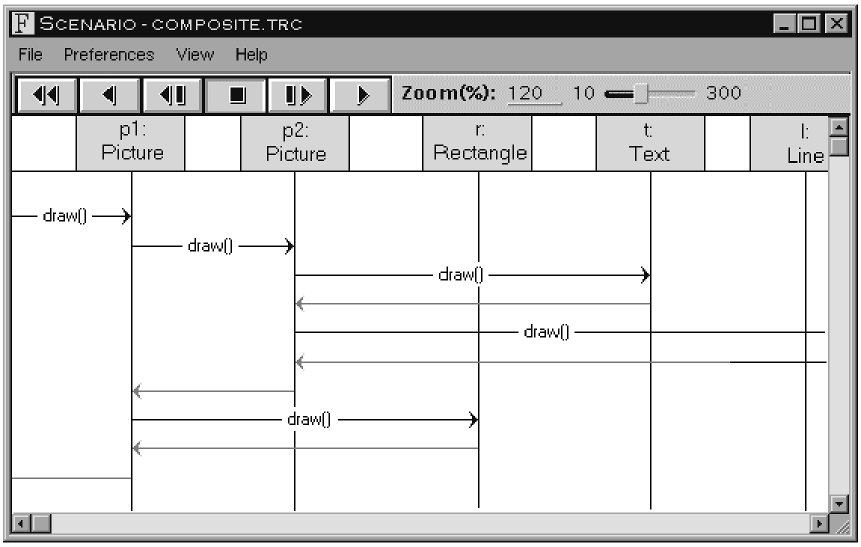

Figure 3.19: *Scenario Diagram*

The user can control the animation with the cassette player like controls in the toolbar. She can rewind the animation statically to examine the trace backwards and forwards (either continuously or step by step). She can also show and hide individual objects and collapse operation call sequences so that only the initial call and its return arrow are displayed. The user can choose not to view any of the operation calls made by a specific object. The whole diagram can also be zoomed to a desired size.

Scenario Diagram has a *File* menu from which the user can open, save, and print animation traces using the usual default dialogs making animation also available for later examination. In the *Preferences* menu the user can control various options including animation delays.

In the *View* menu the user can set filtering and updating options for the window. The features that are subject to the filtering are the visibility of the objects (all objects in the system cannot of course be visible at the same time) and the "openness" of operation calls (a call is open if the calls made inside it are visible). These settings are identical to the ones in *Class Diagram* in that the selections in *Framework Composer* and in *Class Browser* define the default values for filtering. Object- and message-based filtering can be selected directly in *Scenario Diagram*. In particular, the facade object (as described in the *Facade* design pattern) along with all messages sent to and from it can be visible, while the objects in the subsystem behind the facade are hidden unless the user wishes to inspect them in more detail. Another special filtering mechanism can be used to visualize all instances of the same class as a single box representing them all.

Similar run-time behavior visualization tools have been implemented by Koskimies and Mössenböck [KoM96] and Viljamaa [Vil97c].

4. An Example Session

In this chapter we give a small tutorial on how to use the FRED toolset. First we describe how new design patterns can be incorporated into the system (chapter 4.1). After that we show design pattern usage in the realm of framework construction (4.2). Finally we provide an example of framework specialization (4.3).

4.1 Creating New Design Patterns

In our scenario the pattern developer has written or acquired by other means a design pattern that she wishes to utilize in FRED. As an example we use the *Composite* pattern from [GHJ95]. The pattern developer has opened *Pattern Composer* (see figure 3.3) and added the name of the new pattern into the *Available Patterns* list[15]. To categorize the pattern she has given some values between 0 and 1 for every node in the *Categories* tree view. The developer may have also specified new categories by inserting new nodes into the tree view. She has also provided links to some HTML pages so that pattern documentation can be browsed and searched.

The most crucial part of integrating a pattern into FRED is defining its work lists, one for each pattern variation. Through a work list the framework developer gives code templates for the pattern instantiation and provides constraints that must hold for the use of the pattern. Work lists are shown in the *Work List Variants* section.

Work lists are created and maintained by *Work List Editor*. The items are appended into an initially empty work list under a root item that is automatically created when the user first starts to edit the work list. In figure 3.5 the pattern developer has nearly finished the work list for the basic variation of the *Composite* pattern. The items are shown as a tree view. Currently the *Subitems* hierarchy is selected (other hierarchies include, for example, related items and role references). The user has edited the tree by dragging items in the tree view. Most of the items are closed, i.e. their subitems are hidden. The item for the *Composite* participant in the *Composite* design pattern which is currently selected is open, though.

Information concerning the selected item is displayed on the right. The name of the item is simply *Composite*, it is of type *Class*, and it can be used one or more times. A descriptive help text intended for the users of the pattern is also given. This item defines roles for the class name (*Composite*) and for the name of the vector storing references to its children. The restriction that the type really must be *Vector* is given as a constraint clause. There are also separate tab folders for associating a template file and some editor tools with the item.

To define a code template for a work item an existing piece of code is opened in *Template Maker*. In our case (shown in figure 3.6) the pattern developer is making a template for the item describing

[15] All lists and tree views are edited in the obvious way with the *New*, *Change*, and *Delete* buttons. The effect of a button depends on which list is currently active.

Composite.operation participant of the *Composite* pattern. She uses a code fragment from the design pattern's documentation as a basic model for the method. She has marked some tokens to be role definitions (underlined in figure) and references (dashed underlining) to help the pattern user in keeping pattern instances coherent. The marking is done by selecting a text are and associating it with appropriate tags (see appendix B). In the figure the pattern developer is defining the *operationName* role and providing it with a default value.

4.2 Building a Framework

Before building a framework with *Framework Composer* the user must first create a framework project where she defines the name and location (directories) for the project and its files. This is done in the main toolbar's *New Project* dialog (see figure 3.2). In our scenario the user has built her *Draw Tool* framework up to a point where she needs to define the graphic objects handled by the framework. She has already designed classes for some basic shapes such as lines and rectangles, but she also wishes to compose more complex diagrams made up from simpler ones, and what is more important, she wants application developers to be able to define their own shapes. She looks for a solution to her problem by using *Pattern Selection Wizard* (see figure 3.9).

The framework developer focuses her search on *Structural Patterns* and finds some design patterns that could help her. She chooses the *Composite* pattern and pushes the *Instantiate* button. *Framework Composer* then shows a new instance of *Composite* among other patterns she has already applied in her framework (see figure 3.7). The user names the pattern instance as *CompositeGraph* and starts filling in the actors for the roles of the *Composite* pattern. She already has some classes that will play the *Leaf* role in the pattern (*Rectangle* and *Line*). She selects them in *Class Browser*. Then she adds new *Leaf* role place holders (command *Edit* | *New Role...* in the menubar) and associates them with roles by selecting *Edit* | *Associate with Role* in *Class Browser*'s menubar.

To complete the pattern instance the user must produce some new classes to play the *Component* and *Composite* roles. She generates skeletons for the classes using the work list associated with the pattern. She uses *Template Editor* to fill in the blanks. The user generates other classes and their members similarly by applying the work items that are associated with the pattern. After that she validates her pattern instance. The result shows (see figure 3.8) that the system could not find the *Rectangle* class (the user may have accidentally deleted it). To correct the mistake she copies a backup version of the class to its proper directory and then validates the pattern successfully.

The user continues to add code and design patterns to her framework until she regards it to be ready — of course she can always come back to make new modifications[16]. After that she must prepare one or more work lists to guide the application developer to utilize the framework in a desired manner. As a starting point the user has all those work lists that have been produced when instantiating the design patterns in the framework.

---

[16] Note that developing a high-quality framework requires usually many iterations. This causes difficulties because of the dependencies between a framework and applications derived from it (see discussions in chapters 6.2, 6.3, and 6.4).

Some parts of the work lists are not done yet, but rather they describe things the person specializing the framework must take care of (things like creating subclasses and filling in the hot spots). The framework designer selects some of these work items and uses them to produce a new work list that is meant for the framework user. Note that there can be many kinds of typical usage and users for a framework, so there can be also many work lists associated with it. The designer can also create new work items or use some items that she has made before, for example, to guide the documentation of the application that is derived from the framework. In our example the designer is constructing a simple work list for a novice user who wants to derive a minimal application from the framework. The framework developer has already selected and grouped some of her work items (related, for instance, to window creation and graphical objects). Later on she will add some new items and equip all items with help text documents to make the application development a straight forward process.

4.3 Deriving an Application

The framework specialization is handled with *Application Composer*. It processes the necessary framework refinement information in the form of a work list constructed by *Framework Composer*. The work list summarizes the framework developer's idea of how the framework should be used. A typical application developer then follows the selected work list step by step and modifies the arising hot spots to produce a working application.

In our scenario the user wants to build a simple drawing application by using the before-mentioned *Draw Tool* framework (recall chapter 3.3). To achieve this she browses through the available frameworks and work lists with *Framework Browser* (see figure 3.11) and selects the *Draw Tool* framework and its work list called "*Expert work list*". Then the chosen *Draw Tool* framework is specialized with *Application Wizard* (see figure 3.12), where the current work list is shown.

Traversing through the work list is simple; the user selects a task from the work list and follows the instructions shown in the opened editor. The editor view can be *Class Factory* (figure 3.13) which is used to create new subclasses, *Template Editor* (figure 3.14) which is used to modify individual classes, or some *Special Editor*.

During the specialization process each view shows suitable information and examples to help the user. In addition the user can open other documents to get more detailed information if necessary. Documentation and examples has been created by the framework developer. When the user has executed all the items in the work list the application is completed[17] and ready to be compiled with the Java compiler. The framework and all the applications derived from it can be visualized using the *Class Diagram* and the *Scenario Diagram*.

[17] To be more exact, (a minimal version of) the application is ready when the application developer has applied each work item at least as many times as specified by the framework designer in the lower bounds of the cardinality values in the work items. Of course the application developer can further extend her application by applying work items as long as the higher bounds of the cardinality values in the work items are not exceeded.

5. Implementation

In this chapter we take a look at the FRED toolset from the implementor's point of view. First we describe all required concepts (5.1). Then we represent a dataflow diagram and discuss some of the main functions of the system in more detail (5.2). Finally we talk about data storage (5.3) and implementation environment (5.4).

5.1 Basic Concepts

In the following we have formalized all concepts related to the FRED toolset as classes, associations, attributes, and operations. The classes are given as an OMT diagram in figure 5.1 and their detailed descriptions are listed in Appendix C. The attributes, operations, and most of the association names are left out from the figure for simplicity.

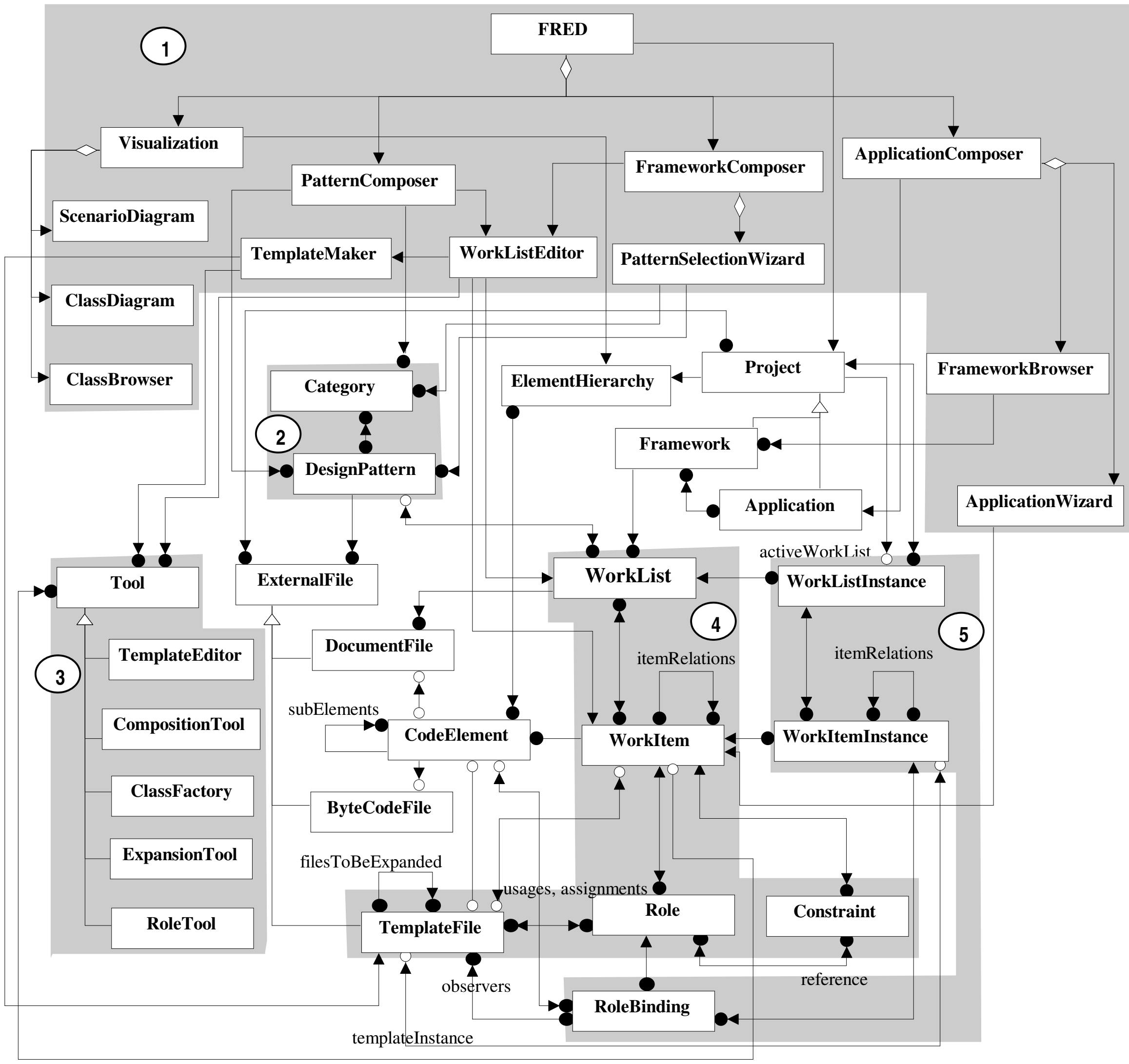

Figure 5.1: OMT class diagram of the FRED toolset

Classes in area 1 are *user interface components*, area 2 includes classes related to *design patterns*, area 3 has classes that represent *special editor tools* for work items, area 4 describes the *work list* concept and classes related to it, and area 5 represents *work lists taken into use*. Other classes included in the diagram are related to *project management*, the Java *code element hierarchy*, and *external files* included in a project.

5.2 Functionality and Dataflow

In this chapter we describe the functionality of the FRED toolset. First we represent a dataflow diagram of the whole system. Then in the following chapters we discuss some of its main functions in more detail.

5.2.1 Dataflow Diagram of the FRED Toolset

Figure 5.2 represents a high-level dataflow diagram of the FRED toolset. There are three logical data repositories: *Design Patterns* (D2) includes formal descriptions of the available design patterns. One pattern description consists of *work lists* (*code templates*, *roles*, and *formal constraints)*, *HTML documents*, and *pattern category information*. *Framework Work Lists* (D3) includes all work list alternatives for every available framework. It also includes *work list instances* (run-time representations of work lists under execution). *Implementation Code* (D4) includes all framework and application code.

The *Pattern Composer* (P1) creates new design patterns. It takes existing design knowledge — namely design patterns (arrow 1) — as its input and asks (2) *Work List Editor* (P2) to create a work list and necessary constraints for the pattern and to associate HTML documents with it. *Work List Editor* calls (3) *Template Maker* (P3) in order to construct code templates for individual work items. *Template Maker* may use existing Java code or old templates (4) as a basis for the new templates. Formalized pattern descriptions are stored (5) into repository D2.

Framework Composer (P4) creates new frameworks by instantiating design patterns and creating proper work lists that guide the application developer in the usage of the framework. When the developer wants to instantiate a design pattern *Framework Composer* asks (6) *Pattern Selection Wizard* (P5) to assist the user in choosing a suitable pattern based on pattern categories and their descriptions. This is helpful as there are dozens of design patterns available, and it might otherwise become laborious to find the right solution to the problem at hand. The available patterns are retrieved (7) from repository D2. *Pattern Selection Wizard* may start *Pattern Catalog* (P6; actually a HTML browser) if the user wishes to study design patterns more closely. It is also possible to make text searches based on the categories or arbitrary HTML documents associated with the currently available patterns.

The chosen design pattern and its work list are returned (6) to *Framework Composer* who makes the initial copy of the work list and stores (10) it into repository D3. The framework developer may use this work list when constructing the final work lists for the application developers. *Framework Composer* can be afterwards used to study instantiated patterns and their relationships with implementation code.

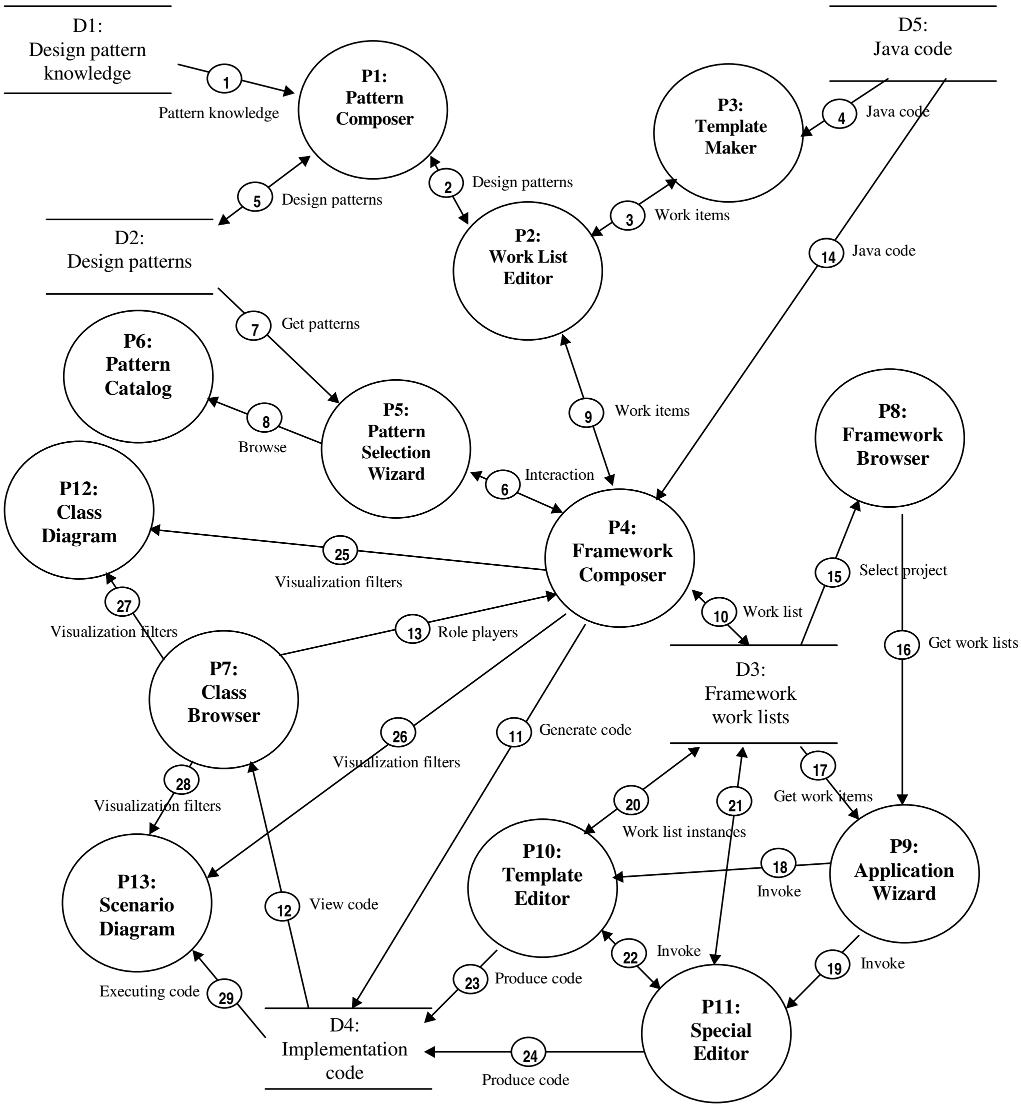

Figure 5.2: Dataflow diagram of the FRED toolset

At this stage it is possible to either directly generate Java code (11) from the pattern or associate existing Java code with the pattern. In the latter case *Class Browser* (P7) is used for examining existing code (12) and choosing Java elements (13) to be associated with some pattern roles. Associations are updated (10) into repository D2. In addition to the work items retrieved from the design patterns the user constructs (14, 9, 3, 4) other work items (possibly concerning code that has nothing to do with patterns) and finally combines the items as the final work lists (10) for the application developer.

The application developer starts by browsing the available frameworks and their work lists (15) with *Framework Browser* (P8). The user chooses one or more frameworks and their work lists (16) and starts executing the work lists with *Application Wizard* (P9). *Application Wizard* fetches (17) the needed work items from repository D3. The execution of one work item means that the user is assisted to fill in missing code fragments for the item with an editor associated with it. *Application Wizard* starts (18, 19) either *Template Editor* (P10) or a *Special Editor* (P11) for editing. *Template Editor* is used as a default but there may also be specialized graphical editors designed just for the purpose.

Both *Template Editor* and *Special Editor* fill in (20, 21) work list instances as the user changes the code. Special editors may also be started (22) from within *Template Editor* as the framework developer might have defined some part of the particular code template to be edited using a special editor. The representations in *Template Editor* and *Special Editor* are kept consistent. As work lists are finished their code templates may be converted (23, 24) into pure Java code[18].

Class Diagram (P12) and *Scenario Diagram* (P13) visualize the application under development. *Class Diagram* represents the application as a static class chart whereas *Scenario Diagram* offers a dynamic visualization of the execution of the application. The design pattern based approach to system visualization used in FRED provides a powerful way to highlight only the information that is currently of interest. Visualizations can be filtered, for example, according to selections from *Framework Composer* (25, 26) or *Class Browser* (27, 28).

5.2.2 Building Design Patterns and Work Lists

In the FRED toolset every design pattern specification consists of work lists (one for each pattern variation), a set of HTML documentation files, and category information. A work list is built by creating a `WorkList` object. It is then initialized with the work list name and documentation provided by the user. After that an appropriate number of `WorkItem` objects is created and they are inserted into the work list. Work items can also be associated with each other in various hierarchies (e.g., dependent items, subitems, and connected items). Whenever a new work item is created it is put straight under the work list (the top level) or under another work item for which it will become a subitem. The created item is initialized with the user specified values for the item's attributes (name, type, template, and so on).

A `WorkList` can be saved as a whole including all its items and connections between them or partially (selected items). In the latter case, only the selected items and connections between them will be saved. Similarly, a work list can be loaded as the new active work list or as a new part for an existing active work list. This makes it possible to combine new work lists from existing ones. The user must specify the location for the new subtree, whether or not the included work list is copied or linked to the existing one, and how the items of the included work list will be associated with the old items.

[18] The conversion may be unnecessary, since all extra information included in template files is coded as tags inside regular Java comments.

When suitable work lists have been constructed for every variation of a certain design pattern the system creates a new `DesignPattern` object. It associates these work lists and the pattern documentation with the created object. The pattern must be also categorized. This is done by connecting the pattern with one or more `Category` objects. Each connection is also assigned a real value to denote how well the pattern fits into that category. New `Category` objects can also be created. After creating a new category some of the old patterns might be connected to that category, too.

5.2.3 Instantiating Design Patterns During Framework Construction

A software project in the FRED environment can either involve creating a new application framework for some (sub)domain or deriving an application from one or more existing frameworks. When the user wants to build a new framework the system first creates a `Framework` object to which the user can gather all existing source code, binary files, and documentation she wishes to use. After that she selects some design patterns she wants to apply in her design. More patterns can be instantiated later as the framework evolves.

Each pattern variation has a `WorkList` that must be followed to implement the pattern. To do that, the system first creates a corresponding `WorkListInstance` object. It stores references to all `WorkItemInstance` objects sorted according to the `WorkItem` object they refer to. The system will also create a minimum number of instances of every `WorkItem` in the list. In principal, the user can browse through the items in the work list and apply them in whichever order she wishes as long as the restrictions posed by the item relations and cardinalities are not violated.

A `WorkItem` is applied by creating a `WorkItemInstance` object which is added to the set of item instances of the active `WorkListInstance`. The user must select whether she wants to create new code to implement the structure that is described by the item or if she wishes to associate existing code for this item and its roles. The minimal unit of separating code generation and code association is thus one work item. In the case of code generation a new copy of the item's template file (and its descriptor object, an object of class `TemplateFile`) is created and it is given a new name. For every `Role` object (i.e. for each role definition) there will be a new "empty" `RoleBinding` object.

The role bindings and the values of the role references in the copied template file are initialized as follows. For every role defined in the instantiated item the `TemplateFile` object of the instance is set as an observer for the `RoleBinding` object and the user must supply the initial value for the binding. Then the system searches through every `TemplateFile` object of all the old `WorkItemInstances` in the active `WorkListInstance` to find the template files that refer to the roles defined in the new `WorkItemInstance`. These `TemplateFiles` are added as observers for the new `TemplateFile`. For the efficiency reasons the system might keep track of references to undefined roles (see also chapter 5.2.4).

When the template file is initialized the user selects a tool from the list of tools specified for the item[19]. The selected tool is invoked with the template file as a parameter. The tool helps the user to fill in missing code fragments to implement the item. A `WorkItemInstance` object is considered done when all its subitems are done and each of its `RoleBinding` objects refer to a name (i.e. when every role has been bound to an existing implementation-level name). The whole work list is considered finished when each of its work items are done.

After the framework is ready (i.e. when the framework developer does not want to apply any new patterns or add code) the developer builds one or more work lists to guide the intended framework users in their specialization effort. All work lists are based on "the root work list" that is automatically created during framework construction. It includes (as its "sub work lists") every work list that has been created when design patterns have been instantiated. The framework developer can take parts of these work lists and manipulate them as she wishes. She can also add new work items to create work lists suitable for various framework applier needs. The created lists are added into the set of work lists associated with the framework.

5.2.4 Template Expansion and Role Binding

`ExpansionTool` takes care of the *intelligent inclusion* of template files into other template files[20]. This feature is required, for instance, when a class is defined as one work item (and template) and its methods are defined each in separate subitems (and subtemplates). Because Java requires all code belonging to a class being in the same file (and inside the class definition), the subtemplates must be "included" into the class template. The place where the subitems will be inserted is marked with appropriate tags in the parent item's template. If there are no such tags specified in the file[21], the user must select suitable location for the expansion.

`ExpansionTool` is invoked by the tool that is currently editing the template file to be expanded. In `TemplateEditor` this happens when the user enters her caret into the section that is surrounded by a tag that refers to `ExpansionTool` (see appendix B for description of tags and the tag language specification). `ExpansionTool` receives as a parameter a name that refers to the `WorkItem` whose instance's template file should be included into the original template file. `ExpansionTool` asks the active `WorkListInstance` for the references to every `WorkItemInstance` that is of the given type. The template files linked with these item instances describe all the template instances (i.e. modified copies of the original template) created so far. The names of these files are then returned to the `TemplateEditor` which expands them at their place. To avoid duplication the expansion is always done so that possible old expanded code is overwritten. Overwriting does not

[19] The default tool is `TemplateEditor`. Other tools include tools for creating new subclasses for existing classes (`ClassFactory`), arranging object compositions (`CompositionTool`), expanding template files with other templates (`ExpansionTool`), and for managing role bindings (`RoleTool`).

[20] Note that this feature is not the same as the "statically defined inclusion" that is part of the tag language specification.

[21] We cannot require that there always exists such a tag. Consider, for instance, a situation where the parent item represents a class and is associated to some existing regular Java class definition without any tags. We must still be able to expand that class definition with some new methods generated from the subitems and their templates.

destroy the user's earlier edits, however, because modifications are always made in the subtemplates. It should be possible to invoke `ExpansionTool` also right after the user has finished editing a subitem.

Role binding is another important mechanism of the FRED toolset that requires working with several templates simultaneously. Role binding is handled by `RoleTool`. It is invoked by the tool that is modifying the template file when the user has given a new value for a certain role. In `TemplateEditor` this means that the user has moved her caret out of the section surrounded by a tool tag referring to `RoleTool` and the string value of the section has been modified (this can be checked by `TemplateEditor` itself or by `RoleTool`, in which case `RoleTool` is invoked always when the caret leaves the above mentioned tag). `RoleTool` gets as parameters a name that identifies the role that is being referred to (e.g., `MyWorkItem.RoleFoo`), the value that the user has just assigned for the role (by editing the section), and the name of the template from which the role was referred to (i.e. the name of the template where the tag was).

`RoleTool` must then find the correct `RoleBinding` object. It first searches through all `WorkItemInstances` that are of given type. After that there are two possibilities depending on the maximum cardinality of the `WorkItem`. If it is *one*, the target of the reference should be clear because the `WorkItem` can have only one instance and thus the role binding can be found unambiguously. If the item instance is already created the binding is fetched and updated. The referring template is added as an observer for the binding (if it was not already). If the item instance is not yet created it is created and initialized as described in the previous chapter.

If the maximum cardinality of the `WorkItem` is *more than one*, the tool must find out if this role reference is already linked with a `WorkItemInstance`. This is done by checking if any of the instances has a role binding (for the given role) with this template as its observer. If the answer is no the user must supply the correct selection from the potential instances (the type of the selection must be the given `WorkItem`)[22]. After the correct instance has been identified (and made observer if necessary) the value for that binding is updated and all observers are notified (and updated subsequently).

5.2.5 Checking Constraints

Constraints are associated with work items when creating patterns using the `WorkListEditor`. We have chosen to represent the constraints for work items as a combination of role references (as in [Sou95]) and predicate logic (as in [FMW97]). The philosophy is that work items (and their role references) are supposed to assist the end-user to build code according to the structure of the item. But, as we do not want to restrict the user's coding unnecessarily, the predicate logic constraints are used in the end to check that the final code corresponds to the initial requirements. Table 5.1 lists the most important predicates. Table 5.2, in turn, lists set operations that can also be used in constraint clauses.

---

[22] The user can also select "none" if, for example, the correct instance does not exist yet.

Constraints are compositions of predicates connected with operators AND, OR, and NOT. Additionally quantors $\forall$ and $\exists$ can be used. The predicates (and constraints composed of them) are implemented in Java using the *Strategy* design pattern. New predicates may be added as needed[23]. For some examples, see appendix A.

Predicate	***Description***
DefinesMethod(aClass, aMethod)	True, if aClass provides an implementation for aMethod.
DeclaresMethod(aClass, aMethod)	True, if aClass provides a signature for aMethod.
DeclaresMethodWithReturnType(aClass, aType)	True, if aClass declares any method with return type aType.
DefinesField(aClass, anotherClass)	True, if aClass defines a field with type anotherClass.
ImplementsInterface(aClass, anInterface)	True, if aClass provides an implementation for anInterface.
ExtendsClass(aClass, anotherClass)	True, if aClass is a direct subclass of anotherClass.
IsDescendant(aClass, anotherClass)	True, if aClass is a descendant (direct or indirect subclass) of anotherClass.
InstantiatesClass(aMethod, aClass)	True, if aMethod creates an object of aClass.
CallsMethod(aMethod, anotherMethod)	True, if aMethod calls anotherMethod.
RefersField(aMethod, aField)	True, if aMethod refers to aField.
IsTypeOf(aField, aType)	True, if aField has type aType.

Table 5.1: Predicates for defining constraints for design patterns

Set operation	***Description***
AllMethods(aClass)	Returns all methods of aClass.
AllFields(aClass)	Returns all fields (data attributes) of aClass.

Table 5.2: Set operations for defining constraints for design patterns

[23] For example, when a new design pattern cannot be described using the existing predicates.

5.2.6 Implementing *Application Composer*

Looking from the *Application Composer* point of view, the stored framework definition needs at least the following components:

- Source code of framework's classes (Java).
- Template files used to instantiate new classes.
- Work lists and their components.
- Documentation, which includes framework descriptions, class descriptions, hot spot descriptions, examples, technical reports, etc.

When an application is being constructed, tasks in the current work list are executed by the application developer. The result will be a set of new classes which can then be combined with the framework classes to obtain the final application. The situation is illustrated in figure 5.3. The original framework code is separated from the application dependent code; this approach avoids unnecessary code duplication and helps maintaining the application as improvements and corrections are implemented to the original framework. Also, a single framework can have multiple work lists associated with it.

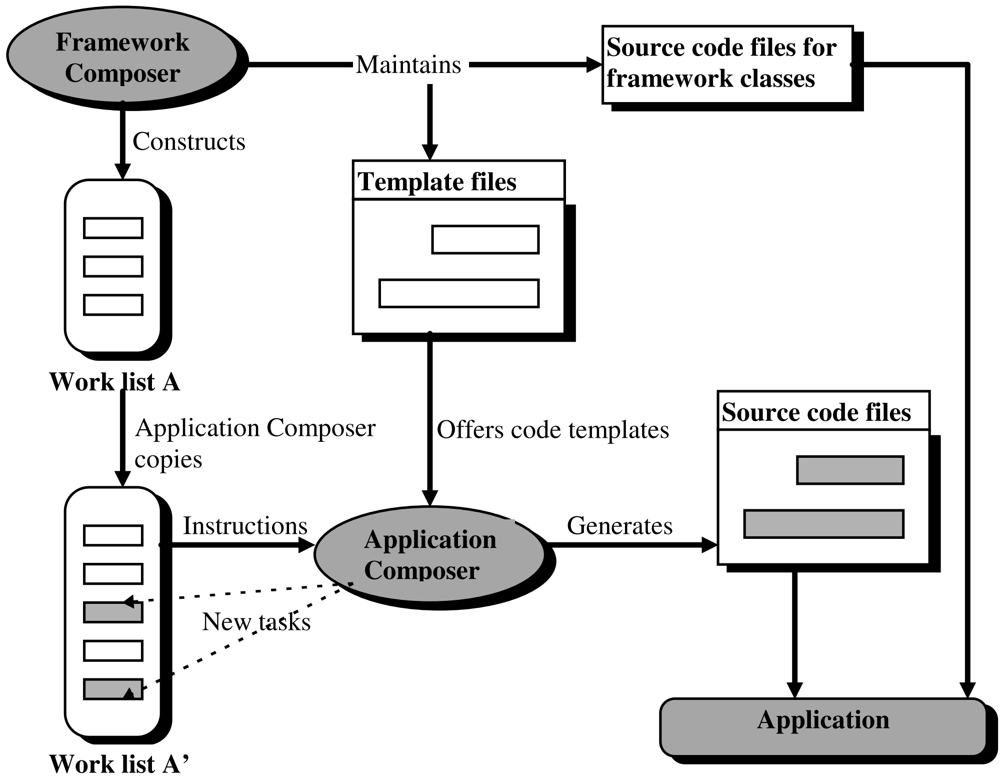

Figure 5.3: Application construction with *Application Composer*

The stored application definition should include the following information:

- A copy of the work list associated with the framework (we need a copy as the work list can be dynamically modified and stored — even across several sessions).
- Generated source code files for application classes.
- A project file which contains, e.g., information about which entry in the work list was last executed and other dynamic information that needs to be stored across sessions.

The final application will be generated by compiling the base framework's classes and the new application classes with the Java compiler. Before the compilation process, *Application Composer* will perform checks for any obvious consistency failures, like undefined hot spots in application classes.

Note that new versions of a framework may not necessarily be compatible with previous ones. A full-blown software version control system would be a very useful tool in monitoring these situations. We also need to provide functionality to check that a modified framework is compatible with an application derived from a previous version of the framework. Of course, such automatic checking cannot ensure full compatibility, but it will be useful as a tool to catch some obvious errors like newly added hot spots not being defined or removed hot spots extraneously being refined in the application.

5.3 External Data Storage

The data maintained by FRED can be classified into the following categories:

1) Java source code (and optional byte code) for the implementation of frameworks and applications.
2) Template files which are used when instantiating Java classes.
3) HTML documentation for patterns, frameworks, work items, etc.
4) Design pattern and design pattern instance definitions.
5) Work list definitions.

The first three items (source code files, templates and documentation) are basically unstructured, sequentially accessed text files. The definitions for design patterns, their instances, and work lists are data with a well-defined logical structure of medium-level complexity containing references to similar and dissimilar data items.

It is clear that the structured data in FRED would benefit from having a readily available distributed object repository, although data storage could be implemented using directly the Java Serialization API. Storing the source code and documentation in an object repository seems a bit excessive as their kind of data does not need the facilities provided by an object repository. Also, as the framework documentation is accessed via a web browser, it needs to be available as plain text.

However, the framework's documentation, source code, design patterns, and work lists form a logical entity with tight relationships between its component parts. When new versions of frameworks and applications are being built, it is clear that we will need to manage all of the data which comprises a framework or application so that documentation, source code, design patterns, and work lists are consistent with each other.

Some basic requirements for the data storage facilities in FRED are:

- Distributed data storage. It is obvious that an application like FRED, where potentially numerous developers are working on a single project, needs to have its data stored in a central repository where it is easily accessible for all users. With *Java*, the distributed approach is also a natural way to approach the data storage, as the language and its support facilities have been developed with emphasis to networked data access.
- Portability across a number of platforms. As the FRED developers will be using at least two platforms for software development (*Sun Solaris* and *Microsoft WindowsNT/95*) and *Java* itself is easily portable across different hardware/software environments, the choice of data storage implementation should not restrict the development and usage of FRED in these environments.

Interfacing relational databases with JDBC

JDBC is the *Java* interface to database systems (typically relational databases) accessible via the *SQL* query language. The *JDBC API* provides access to numerous commercially available *SQL*-accessible databases, e.g., all databases which support access via *ODBC* (*Open Object Database Connectivity*), can be used via *JDBC* via the *JDBC-ODBC* bridge. Mapping of the *Java* objects to rows of tables in a relational database is done in code written by the application developer. Creating this mapping is typically straightforward, but incurs a small development time overhead.

We have looked at a number of different data storage technologies considering them as candidates for storing FRED persistent data. For a number of reasons we have decided to use *JDBC* to interface with relational database management systems to store the structured data within FRED. Some of the reasons for selecting *JDBC* as access method are enumerated here:

- Relational databases as commonly used in the industry and FRED can easily be deployed for testing and evaluation to receive feedback during the development period.
- FRED's persistent data structures can be made compatible with the *Stone AF* repository with relatively modest development effort.
- The development team does not need to acquire additional — potentially expensive — development tools as we will have available *Borland*'s *Interbase DBMS* as part of the *JBuilder* development environment for *Java*.

We also considered other means for storing and accessing the FRED persistent data. While they also are viable implementation methods, they would complicate the deployment of the FRED system as additional database access components would need to be acquired and included with FRED. Using *CORBA* and Object databases is shortly considered in the following chapters.

Interfacing databases with CORBA

Object Management Group's (OMG) *Common Object Request Broker Architecture* (*CORBA*) — see `http://www.omg.org` — defines a set of facilities for accessing objects across networks using *Object Request Broker*s (*ORB*). *ORB*s act as mediators between a client and a server, finding the requested object, invoking object's methods, converting between data types, etc. The remote objects can be implemented in various programming languages, e.g., *C++* and *Java*. To access remote objects, their functionality and visible structure is specified via *IDL* (*Interface Definition Language*). Among other things, *IDL* enables one to, e.g., inherit *Java* objects from *C++* objects. Mappings from *IDL* definitions to actual programming language objects are provided by *IDL* compilers.

Interfacing with *Stone AF* from FRED would necessitate:

- Obtaining a CORBA-compliant set of tools for development (e.g., *OrbixWeb* by *IONA* (`http://www.iona.com`), *CORBAplus Java* edition by *ExperSoft* (`http://www.expersoft.com`) and *InterLanguage Unification System (ILU)* by *Xerox PARC* (`ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`).
- Creating *IDL* definitions for *AF* objects.

Currently, CORBA offers the only widely accepted standard for doing distributed computing in a multi-language implementation environment. While CORBA cannot really be described as a simple, easy-to-learn development environment, CORBA-based solutions have already been tried and tested in the field.

Interfacing Object Database Systems

There are a number of ODBMSs (*Object Database Management System*) available which provide direct storage of *Java* objects across networks, e.g., *ObjectStore* by *Object Design Inc.* (*ODI*, `http://www.odi.com`), *Versant* ODBMS by *Versant Object Technology* (`http://www.versant.com`). *ODI* also has a freely available *ObjectStore* compatible single-user, single database *ODBMS*, *ObjectStore PSE for Java*. From the developer's point of view, object database systems probably provide the simplest means of storing and accessing remote *Java* objects. Especially the phase of defining the data for the DBMS is usually very simple or even non-existent.

FRED database implementation

The logical structure of a FRED database will be based on an expanded version of the database structure used in *Stone AF*. Some of FRED objects are already modeled in *Stone AF*, like classes and diagram items. A number of objects — design patterns, work lists — will need to be modeled and implemented as *Stone AF* compatible tables. The underlying database storage structure implementation will be transparent to the application code at large.

5.4 Environment and Libraries

We consider Java to be an excellent language for implementing frameworks and framework tools, because it is safe, it has clean and simple object-oriented structures, and it assures portability and flexibility of systems [Vil97a]. That is why the FRED toolset will be implemented using the Java language (and namely the JBuilder [Bor97] environment). For more details on our implementation environment see [HTV97]. We intend to use the Java coding standard defined in [AMB97].

The internal representation of Java source code of a framework (or an application) under development is constructed using the Java Reflection API [Sun97a] or by directly parsing the source code. There are many freely available parsers and parser generators for Java (see, for example, Sun's Java Compiler Compiler [Sun97b]). The FRED tools also access source code files so that they can be shown, edited, and (re)compiled. We use the standard JDK 1.1 compiler and interpreter. Other interfaces include support for various output formats such as PostScript [Lan91], and interfacing with version control systems such as RCS [Tic85] (see also chapters 6.2 and 6.3).

Graphics is needed in the FRED toolset (at least) for representing charts in *Class Diagram* and *Scenario Diagram* windows, as well as when drawing class and other relationship trees and graphs, for example in *Class Browser*. The graphics will be implemented using the JFC (Java Foundation Classes) library. Controlling the application's dynamic behavior when tracing and visualizing its execution can be implemented using Java threads. *Template Editor* will be derived from a text editor framework that is being developed within the project [Hak97].

6. Software Development Environment

In this chapter we describe the FRED tool set's planned support for team work (6.1), how FRED can be used with Java Programming enviroments (6.2), and the role of version control and configuration management in FRED (6.3, 6.4). We will discuss the evolution of frameworks and applications derived from them (6.5). Finally, we will present some features which are not yet firmly placed into FRED implementation schedule, but would be useful to have in a FRED-like development tool (6.6).

6.1 Team Development

Optionally, FRED should provide support for several developers working simultaneously on a framework or an application. This is because a development environment based on design patterns and frameworks is potentially used to develop large frameworks and applications involving numerous software developers. Such multi-developer support would necessitate providing at least the following features:

- Allowing several developers to work simultaneously with different parts of a framework or an application, so that one modifiable entity can be edited by a single developer at a time.
- Supporting locking of several entities for modification at the same time or even locking the whole project for exclusive use.
- Providing a user catalog so that a developer can see who has what parts of the system reserved for modification at certain time.

Supporting an arbitrarily large development organization would probably also oblige to have a central repository for storing the design pattern library, the framework implementations, and possibly also the specialized applications.

6.2 FRED and Software Development Environments

FRED can easily be used with Java code development environments like Borland JBuilder. As the Java code maintained within FRED is stored in standard text files, a Java compiler and debugger can easily access the source code of an application created with FRED. As the tags of the template editor are embedded inside Java comments, they do not interfere with any external tool requiring access to the source code. We have not studied in detail how FRED could be integrated more tightly in a integrated development environment (IDE) like JBuilder.

Especially building a user interface with a tool like JBuilder would be a sensible thing to do, as FRED does not provide any special support for that kind of activity. Typically, the debugging of an application would also take place within a Java IDE or — alternatively — using the JDK Java Debugger (jdb).

It is also conceivable that one would use a Java editor provided by a Java IDE for creating some portion of code for a particular framework or an application. One could, e.g., build a design pattern's initial body with FRED, edit it with an IDE Java editor, and then use it as such within FRED. However, as FRED and the IDE are not integrated and aware of each other, it is possible that the user would, e.g., delete some tags created by FRED and the resulting template would be incorrect. Preventing this kind of situations will require verifying the contents of Java code files within FRED. Currently, while using an external editor for Java files is possible, we do not provide any special support for it within FRED.

6.3 Version Control of Frameworks and Applications

During the lifetime of a framework it is almost inevitable that the framework will need to be modified and improved based on the experience gained during the application development and maintenance. Some of these changes to a framework will force to modify the applications based on the framework (e.g., changes to the inheritance structure of component classes), while some other changes will not necessarily require modifications to the applications (e.g., changing the functionality of some component classes). The most significant data dependencies in FRED repository are depicted in figure 6.1.

It seems obvious that modifications in certain data will potentially lead to changes in its dependent data. Changes to the design patterns are probably relatively rare, as design patterns have typically emerged during a long period of development of various software systems. However, changes to a design pattern potentially propagate modifications all the way down to the application level. In practice, the more typical change location (beside the application code) is the framework level, but it would be natural and consistent to apply version control to design patterns and their instances as well.

As mentioned earlier, modifying a framework potentially requires a number of changes to those applications which are based on that particular framework. It is possible that several applications have been based on that framework and it may not be feasible to update all of them simultaneously. Thus, we would like to have the original framework available as well as its modified versions which will potentially be used as a base for further application development. We could then upgrade the older applications to the new framework version as time permits.

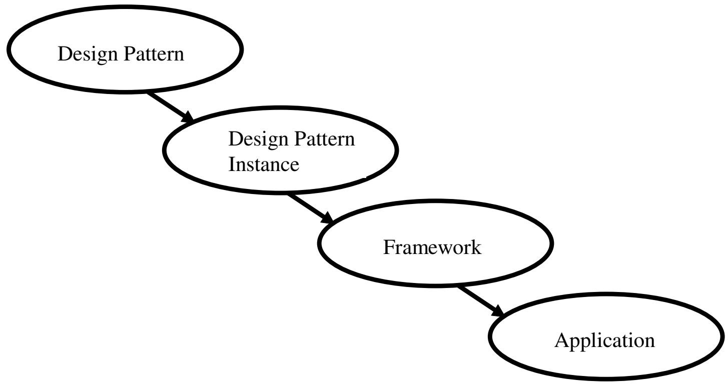

Figure 6.1: Data dependency levels in FRED

To aid the user in propagating upper level changes to the lower levels of figure 6.1, it would be convenient if FRED could perform some tasks automatically, such as:

- Determine if a change will potentially or necessarily induce changes on other levels, i.e., if the dependents are still consistent with the ancestor after it has been modified.
- Bookmark the change locations so that they can be visited later by the user.
- Perform or suggest the necessary changes automatically.

The developer of an application must have a stable foundation to build on: while she is building the application, nobody should be simultaneously making modifications to the underlying framework or to the components the framework depends on. This implies that either the application developer can lock the ancestor tree from modifications (could be done implicitly by FRED) or any modifications to ancestor components will always create a new version of those particular components, thus leaving the existing versions of components available for continuing use.

In effect, we need a mechanism where a FRED-maintained building block which has dependents cannot be modified by default, but instead a new, modified generation (version) will be created. And, as discussed earlier, we also need a mechanism to aid the framework and application developers to propagate these modifications back to the dependent “lower-level” components.

Implementing support for full-fledged version control is quite an effort and outside of the main research project topic. Thus, it would probably be best to design an abstract version control interface which then could be specialized for some actual standard version control systems. We could then even begin FRED development by having no version control system available at all. However, all accesses to the data stored by FRED would be made as if a version control system were present.

6.4 Interfacing Version Control Systems

An application developer is not typically allowed to modify any other data than those which are concerned with the specialization and implementation of the actual application, i.e., framework definitions and contents of the design pattern library are available to her as read-only data only. Thus, an application project consists of the work list associated with the base framework and of the application specialization information (refined hot spots, newly created classes for the application, etc.). If FRED is prepared to undo every user action, we must also maintain — perhaps even across multiple sessions — a full log of executed actions (both user-activated and auto-activated) to be able to undo them.

The interface to a version control system can be relatively simple: basically we need to be able to lock components under modification. Based on what components the user will reserve for editing, FRED will internally make reservations for other components which are dependent on the user-requested components, when necessary. This implies that we need to have access to the version control system's features via a callable API or a command line interface which can be executed from within FRED.

6.5 Creating Partially Specialized Frameworks

The intended usage mode of the FRED toolset is to create ready-to-run applications based on mature frameworks. The possibility of specializing another, refined framework from an existing one (see figure 6.2) has been considered by Koskimies and Mössenböck [KoM95]. They have described a two-phase approach to framework design where a specific application is generalized in successive steps to obtain the most general form of the application specification. The successive generalizations are then implemented as frameworks starting at the most general level. Another situation, where support for partially specialized frameworks would be useful, is the evolutionary case where — perhaps after using the framework to implement several applications — it is perceived that creating a specialized framework for a certain application subdomain would be advantageous.

The case where a base framework (*FW*) is specialized to form a partially specialized framework (*F0*) for a certain subdomain could be already realized and accounted for when designing the base framework. However, it is more typical that the need for a partially specialized framework will be detected during the lifetime of the framework and while it is being used as a basis for increasingly more applications in a certain subdomain (*AD0, AD1, and AD2*). In our example we have two applications (*AD0, AD1*) within a particular subdomain and there exists a need for a third one (*AD2*). All these three applications are assumed to have a number of common features.

However, the framework (*FW*) has already been — or will be — used as a basis for applications (*Ax*) of a somewhat different nature. This means that we need to retain the capability of keeping the base framework separate and be able to use it as a basis for further application specialization (*Ax*).

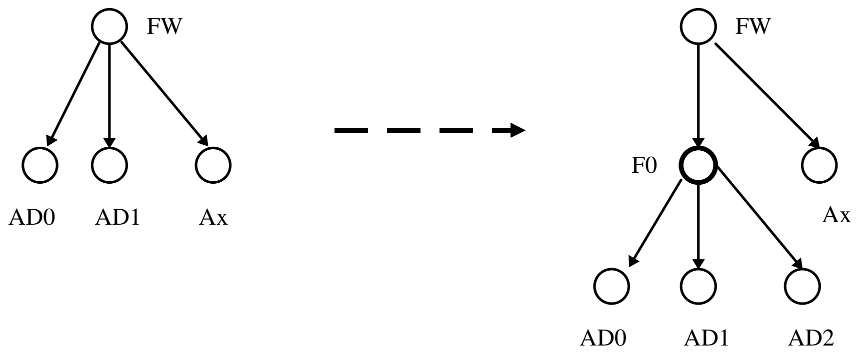

Figure 6.2: Framework hierarchy restructuring

This process of framework/application hierarchy restructuring seems to have a lot in common with the process of class hierarchy restructuring during its lifetime (see, for example, [Cas92]). The main difference is that with frameworks, we have a larger scale: we inherit and specialize groups of classes instead of single classes.

It would be relatively straightforward to extend FRED to support the case where a base framework is used for creating a number of partially specialized frameworks and these frameworks would in turn be used as basis for new, complete applications. In this case we would define where and how the partially specialized framework will be stored so that the information contained in the base framework definition will not be unnecessarily — and preferably not at all — duplicated in the definition of the partially specialized framework.

As part of the support for partially specialized frameworks, FRED would need to provide the functionality to combine the definition of the base and partially specialized frameworks, so as to give a uniform view over specialized frameworks. Of course, the framework specialization process can be carried on to arbitrary depths (hierarchy levels).

The restructuring of a framework hierarchy presents interesting opportunities for automatic tool support. A tool could, e.g., analyze a group of applications and suggest the "best" way of organizing the framework/application hierarchy. The tool could also optimize the hierarchy automatically or interactively guided by the user.

We could decide that FRED will provide no special support at all for restructuring of framework/application hierarchies, thus leaving the burden of building a partially specialized framework and analyzing, verifying, and editing already built applications entirely to the user. With large frameworks and applications this restructuring process is potentially a tedious and time-consuming process which could greatly benefit from any help provided by automatic analysis and editing tools.

Note that we should also be able to use the *Framework Composer* part of FRED together with *Application Composer* while specializing new frameworks by using design patterns to create new classes. This implies that they are tightly integrated: the framework developer, for example, will need to be able to use *Application Composer* when creating new versions of a framework.

7. Related Work

The need for tool support for design pattern -based software development has been widely recognized. At the moment, however, there are just a few such tools available. In this chapter we briefly describe some of the more interesting ones and discuss their relation with our FRED tools.

In [BFV96] Budinsky *et al.* describe a tool that automates the implementation of design patterns using WWW as a medium to access design knowledge and to gather application-specific information. Their tool incorporates an on-line *hypertext* rendition of [GHJ95]. The user can conveniently access the material, follow links to related subjects, and search for information about various topics. In addition there is a *code generation* facility associated with every pattern. The user enters the application specific names and selects trade-offs. Then the tool generates the corresponding C++ declarations, implementations, and even a running demo program. Other similar tools include Designer's Assistant [ACL97] and SNIP [Wil96].

All above mentioned tools use variations of *template-based macro expansion mechanisms* to produce target language code. In all of them the generated code is integrated into the application by the "cut and paste" method, which implies *invasiveness* and *irreversibility* [BFV96]. The user must know exactly what to cut and where to paste it, something that is not always obvious. Once generated code is incorporated into the application, any changes that involve regeneration of code will force the user to reincorporate the pattern code into the application by hand.

Meijler *et al.* [MDE97] call this problem the *design – implementation gap*: changing generated code breaks the connection between design patterns and the implementation. As a solution they introduce an environment (*FACE*) based on "explicit representation of design patterns". In FACE [MED97] the framework developer creates a *metaschema* that describes the syntax rules for the pattern — all the roles and their relationships. Metaschema acts as a set of constraints that is checked when the application developer instantiates the pattern and adds new components included in the pattern. Instantiation is done visually. Metaschema remains at the background during the execution. As a starting point for the application developer the system generates a *primalschema* from the metaschema. Primalschema includes the abstract classes and their relationships. All components in the schema are represented visually and may be directly edited. New relationships are constructed by dragging links between components. The system checks from the metaschema whether the relationships are appropriate or not.

The drawbacks in FACE are those met already in the early 4G languages: in reality the user wants to be able to edit the source code directly — not always through some high-level graphical interface. This is also a prerequisite for using existing code. Our *Template Editor* tries to combine benefits of visual and textual editing. It has been influenced by the idea of enhanced hypertext or *"active text"* (see, e.g., [Con87], [MöK95]). We think that providing step by step instructions and code templates in which some parts of text can be associated with high-level graphical tools and documentation is an elegant way to assist application developers.

FACE, in turn, does not guide the specialization process of a framework. In order to know what kind of components to add to the primalschema the framework specializer must be familiar with the design patterns. The metacomponents in FACE describe (in some detail) the allowed specializations, but the system is only able to warn the user if she tries to do something wrong. We believe that deriving an application from a framework is such a complex and diverse process that it must be explicitly and automatically guided. This is just what our *work list* representation does.

In their article Meijler *et al.* only deal with individual design patterns and predict that FACE works nicely also with a large set of combined design patterns (or a framework). However, a real world framework not only includes complex combinations of design patterns, but also code that does not necessarily belong to any design pattern. Also the pattern description mechanism utilized in FACE seems quite complex. Do we always need new special primitives in every new pattern and how do we introduce more of them? In FRED we always describe the patterns using similar work list representation. If the available set of constraint predicates is not sufficient enough, the pattern developer may introduce new predicates. Work items may also involve code that is not included in any pattern.

All in all, our solutions to the design – implementation gap is quite similar to the one applied in FACE. We use proper internal representations for design pattern instances, to fully integrate them with the internal representation of the implementation level entities, and to implement multiple level undo for all system's functions.

Soukup takes a completely different approach. He proposes *integrating patterns into code* [Sou95]. Soukup's technique includes heavy usage of C++'s preprocessor primitives, which is not a type-safe nor a very clean way to implement macro expansion semantics. Others (see [GiL97], for example) have proposed that design patterns should be promoted as *new language features* in the future. Some attempts to this direction have already been made (see, e.g., [Duc97], [MJR97], and [AgC97]). This could be a better way to encourage pattern usage, but we have decided to avoid designing yet another new language. Instead, we have concentrated on developing new software engineering tools to be used with existing languages, such as Java, which seems to become the new industry standard programming language.

Besides generating code from the design patterns, it is important to keep them *consistent* with the implementation. Kim and Benner have developed the POE (Pattern-Oriented Environment) tool for creating, deleting, and verifying pattern instances and their role mappings [KiB96]. The tool implements *validation algorithms* to ensure that different pattern instances and role bindings are used properly. Algorithms perform graph traversals, and check components and their attributes for possible conflicts. Sefika *et al.* have implemented a similar tool, called Pattern-Lint, for confirming that implementation corresponds to the expected design models, such as a set of design patterns [SSC96a]. Pattern-Lint incorporates also dynamic visualization with static analysis.

In [FMW97] Florijn, Meijers, and van Winsen propose an interesting tool based on *fragments* that represent design elements (design patterns, charts), implementation entities (classes, methods), or other objects (comments, arbitrary text). The system being developed is represented as a graph of interrelated fragments of different types. Fragments can also have behavior associated with them, such as checking consistency constraints or implementing pattern-specific operations. The fragment model is based on *cloning*. Fragments representing design patterns, for example, act as templates for design pattern instances that are created by copying the original pattern fragments and by associating their roles with other fragments that represent implementation-level entities.

We have adopted a similar approach in the FRED toolset. Our *work items* and their template representations come close to Florijn *et al.*'s fragment concept. Their tool, however, relies heavily on the dynamic nature of their implementation platform — the Smalltalk system — and it provides little support for code generation and none for the guided use of frameworks and the visualization of dynamic framework features. In the FRED project we try to show that a similar tool can be implemented also in a statically typed language such as Java. We also aim at developing more intuitive and easy-to-use user interfaces for our tools.

The idea of a work list is also similar to the *application experts* or *wizards* used in some programming environments. A single work item is a kind of a *hook*, discussed by Froehlich *et al.* in [FHL97]. Hooks are framework documentation items that concentrate on how the framework is intended to be used. Individual hooks provide solutions to focused problems, whereas complex problems can be solved by groups of hooks. They have also noticed that the use of individual hooks may be partially automated; the developer selects a hook and the system assists through the use of it. Our work items are more concrete than hooks and as design patterns (elements of frameworks) are mainly described as work items, most of the work items needed for the whole framework are obtained straight from the used patterns. This makes constructing work lists a straight forward process. Note that work lists and their role bindings also provide a method for coupling design patterns with the actual implementation.

A difficult problem in developing object-oriented software is keeping track of the complicated object structures during system execution. In [KoM96] Koskimies and Mössenböck present the Scene tool that integrates *animated scenarios* with documentation and source code. Other such execution tracing tools are described, for example, in [LaN95] and [SSC96a]. The problem with these tools is the enormous amount of information they produce. Typically a realistic system will bring about thousands of operation invocations and it is generally impossible to represent them all graphically without some kind of *filtering*. Design patterns in the FRED toolset provide a natural way to focus the user's attention to the interesting parts of the system and to hide all unnecessary details.

8. Additional Features for Future Consideration

Some features and functionality which will be researched in more detail in the future and implemented in FRED if time and resources permit, are listed here:

- *Pattern mining*. Automatic searching of design patterns from existing source code.
- *Anti-pattern mining*. Automatic searching of non-desirable or faulty coding practices from source code.
- *Framework aggregation and decomposition*. Assembling of smaller frameworks into larger ones and vice versa.
- *Automatized pattern selection*. A tool — *Pattern Selection Wizard* — would find an appropriate pattern for a specific purpose based on problem description given by the user.
- *Consistency checks*. E.g., validating whether an imported Java class fits in a particular role of a pattern. Also, when the user modifies a Java class file — belonging to a project — outside FRED, we could analyze the modified code and attempt to determine whether it is still consistent with the roles it is in within the project.
- *Visualization of frameworks*. Helping the developer to gain insight in the framework's overall structure which may be difficult especially with large frameworks. E.g., scenarios could be used for gaining insight into dynamic behavior of design patterns.
- *Sophisticated composition tool*. We are planning to enhance the graphical composition tool based on FRED user feedback which will be much needed when refining the most useful features of the tool
- *Framework and application testing*. Versatile support for testing of frameworks and applications specialized from them.
- *Automatic framework documentation*. Existing framework structures, e.g., work lists and design patterns, can be used to generate framework documentation.
- *Change management*. Full implementation of source code and configuration management.

These features have not been fully evaluated with respect to feasibility of implementation in FRED, i.e., whether we will have the necessary resources for implementation or would implementation of a particular feature be a reasonable thing to do considering the required implementation effort.

References

ACL97 Alencar P., Cowan D., Lichtner K., Lucena C., Nova L., Tool Support for Design Patterns. Internet: `ftp://csg.uwaterloo.ca/pub/ADV/theory/fmsp96.ps.gz`.

AgC97 Agerbo E., Cornils A., Implementing GoF Design Patterns in BETA. In: Bosch J., Hedin G., Koskimies K. (eds.), *Proc. LSDF'97 Workshop on Language Support for Design Patterns and Object-Oriented Frameworks*. Research Report 6/97, University of Karlskrona/Ronneby, Department of Computer Science, 1997. Internet: `http://www.ide.hk-r.se/~bosch/lsdforg/cornils.ps`.

AMB97 Ambler S., AmbySoft Inc., Java Coding Standard. Internet: `http://www.ambysoft.com/javaCodingStandards.html`, 1997.

BFV96 Budinsky F., Finnie M., Vlissides J., Yu P., Automatic Code Generation from Design Patterns. *IBM Systems Journal* 35, 2, 1996, 151-171.

Bor97 Borland Inc., JBuilder HomePage. Internet: `http://www.borland.com/jbuilder`, 1997.

Bro96 Brown K., Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Internet: `http://www.ksccary.com/thesis/thesis.htm`, 1996.

Cas92 Casais E., An Incremental Class Reorganization Approach. In: *Proc. ECOOP '92 European Conference on Object-Oriented Programming*, Utrecht, The Netherlands, June/July 1992, Lecture Notes in Computer Science 615, Springer-Verlag, 114-132.

Con87 Conklin J., Hypertext: An Introduction and Survey. *IEEE Computer 20*, 9, 1987, 17-41.

CoS95 Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*. Addison-Wesley, 1995.

Deu89 Deutsch L., Design Reuse and Frameworks in the Smalltalk-80 System. In: Biggerstaff T., Perlis A. (eds.), *Software Reusability Vol. II*, ACM Press 1989, 57-71.

Duc97 Ducasse S., Message Passing Abstractions as Elementary Bricks for Design Pattern Implementation. In: Bosch J., Hedin G., Koskimies K. (eds.), *Proc. LSDF'97 Workshop on Language Support for Design Patterns and Object-Oriented Frameworks*. Research Report 6/97, University of Karlskrona/Ronneby, Department of Computer Science, 1997. Internet: `http://www.ide.hk-r.se/~bosch/lsdforg/ducasse.ps`.

FHL97 Froehlich G., Hoover H., Liu L., Sorenson P., Hooking into Object-Oriented Application Frameworks. In: *Proc. ICSE'97, International Conference on Software Engineering*, Boston, May 1997, 491-501.

FMW97 Florijn G., Meijers M., van Winsen P., Tool Support for Object-Oriented Patterns. Utrecht University, Department of Computer Science, the Netherlands. Internet: `http://www.cs.ruu.nl/~florijn/papers/pattern-tool-overview.ps.gz`, 1997.

GHJ95 Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

GiL97 Gil J., Lorenz D., Design Patterns vs. Language Design. In: Bosch J., Hedin G., Koskimies K. (eds.), *Proc. LSDF'97 Workshop on Language Support for Design Patterns and Object-Oriented Frameworks*. Research Report 6/97, University of Karlskrona/Ronneby, Department of Computer Science, 1997. Internet: `http://www.ide.hk-r.se/~bosch/lsdforg/lorenz.ps`.

Hak97 Hakala M., Java Framework to Format, Represent and Edit Active Rich Text Documents. Report B-1997-X, Department of Computer Science, University of Tampere, 1997. (To be published.)

Hau97 Hautamäki J., A Survey of Frameworks, Report A-1997-3, University of Tampere, Department of Computer Science, 1997.

HTV97 Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J., FRED-projektin työvälinevalinnoista. Internal Report, University of Tampere, 1997.

Joh92 Johnson R., Documenting Frameworks with Patterns. In: *Proc. OOPSLA '92, ACM SIGPLAN Notices* 27, 10, 1992, 63-76.

Joh94 R. Johnson, Documenting Frameworks. In: Frameworks Digest, vol. 1, no. 13, Oct. 26, 1994. Internet: `ftp://st.cs.uiuc.edu/pub/FWList/v1n13`.

KiB96 Kim J., Benner K., An Experience Using Design Patterns: Lessons Learned and Tool Support. *Theory and Practice of Object Systems (TAPOS)* 2, 1, 1996, 61-74.

KoM95 Koskimies, K., Mössenböck, H., Designing a Framework by Stepwise Generalization. In: *Proc. 5th European Software Engineering Conference (ESEC '95)*, Sitges, Spain. Lecture Notes in Computer Science 989, Springer 1995, 479-498.

KoM96 Koskimies K., Mössenböck H., Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In: *Proc. 18th IEEE International Conference on Software Engineering*, Berlin 1996, IEEE Computer Society Press 1996, 366-375.

Lan91 Lancaster D., What Is PostScript: A Tutorial. *ACM SIGForth Newsletter 3*, 2, 1991, 15-19.

LaN95 Lange D., Nakamura Y., Interactive Visualization of Design Patterns Can Help in Framework Understanding. In: *Proc. OOPSLA '95, ACM SIGPLAN Notices* 30, 10, 1995, 342-357.

Lew95 Lewis T., et al., *Object-Oriented Application Frameworks*. Manning, 1995.

MCK97 Meusel M., Czarnecki K., Köpf W., A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. In: *Proc. ECOOP'97, 11th European Conference of Object-Oriented Programming*, Jyväskylä, Finland, June 1997.

MDE97 Meijler T., Demeyer S., Engel R., Making Design Patterns Explicit in FACE — A Framework Adaptive Composition Environment. In: *Proc. 6th European Software Engineering Conference*, Zurich, Switzerland, September 1997, Lecture Notes in Computer Science 1301, Springer-Verlag, 1997, 94-110.

MJR97 Matthijs F., Joosen W., Robben B., Vanhaute B., Verbaeten P., Multi-level patterns. In: Bosch J., Hedin G., Koskimies K. (eds.), *Proc. LSDF'97 Workshop on Language Support for Design Patterns and Object-Oriented Frameworks*. Research Report 6/97, University of Karlskrona/Ronneby, Department of Computer Science, 1997. Internet: `http://www.ide.hk-r.se/~bosch/lsdforg/matthijs.ps`.

MöK95 Mössenböck H., Koskimies K., Active Text for Structuring and Understanding Source Code. Johannes Kepler Universität Linz, Institut für Informatik (Systemsoftware), Report 3, August 1995.

Pre95 Pree W., *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

Sou95 Soukup J., Implementing Patterns. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*. Addison-Wesley, 1995, 395-412.

SSC96a Sefika M., Sane A., Campbell R., Architecture-Oriented Visualization. In: *Proc. OOPSLA '96*, ACM SIGPLAN Notices 31, 10, 1996, 389-405.

SSC96b Sefika M., Sane A., Campbell R., Monitoring Compliance of a Software System with Its High-Level Design Models. In: *Proc. 18th IEEE International Conference on Software Engineering*, Berlin 1996, IEEE Computer Society Press 1996, 387-396.

Sun97a Sun Microsystems, Inc., JDK 1.1.3 Documentation. Internet: `http://java.sun.com/products/jdk/1.1/docs/index.html`, 1997.

Sun97b Sun Microsystems, Inc., Sun Compiler Compiler (JavaCC) — The Java Parser Generator. Internet: `http:// www.suntest.com/JavaCC/`, 1997.

Tic85 Tichy W., RCS: A System for Version Control. *Software — Practice and Experience 15*, 7, 1985, 637-654.

Vil97a Viljamaa A., Application Frameworks in the Java Environment. Report C-1997-24, University of Helsinki, Department of Computer Science, 1997.

Vil97b Viljamaa A., Tietoliikenneprotokollien graafinen simulointi. Master's Thesis, Report C-1997-4, University of Helsinki, Department of Computer Science, 1997.

Vil97c Viljamaa J., Tools Supporting the Use of Design Patterns in Frameworks. Report C-1997-25, University of Helsinki, Department of Computer Science, 1997.

Wil96 Wild F., Instantiating Code Patterns — Patterns Applied to Software Development. *Dr. Dobb's Journal 21*, 6, 1996, 72-76.

Appendix A: Example Work Items and Constraints

As examples of defining work items and constraints for design patterns we give definitions for *Composite*, *Abstract Factory*, and *Prototype*. Note that these are formalizations of the basic versions of the patterns as described in [GHJ95]. No variant implementation structures are considered.

For each work item we have listed its attributes, constraints, and template file (if it has one). The constraint specifications are represented as described in chapter 5.2.5. The template files are represented as Java code, but we have used the following short hand notations for the tag language comments (see table A.1). We have omitted some tags and item attributes — such as those representing help texts and controlling editability of text areas.

Short hand notation	**Actual tag notation**	**Meaning**
<def> myRoleName </def>	/*! BEGIN HOTSPOT OUT = FRED.RoleTool.definition(roleName = "myRoleName") */ <Current value of the role> /*! END HOTSPOT */	A role definition. If the user changes the value of the role, the new value is assigned to every code fragment that refers to this role.
<ref> myRoleName </ref>	/*! BEGIN HOTSPOT OUT = FRED.RoleTool.reference(roleName = "myRoleName") */ <Current value of the role> /*! END HOTSPOT */	A reference to a role definition. If the user changes the value of the role, the new value is assigned to every role that is referring to the same role definition.
<expansion> myItemName </expansion>	/*! BEGIN HOTSPOT IN = FRED.ExpansionTool(itemName = "myItemName") */ <Expanded text> /*! END HOTSPOT */	A reference to a work item whose template file is to be expanded here.

Table A.1: Short hand notations used in the template descriptions

Composite

The *Composite* pattern consists of the following work items (figure A.1). *CompositePattern* combines the upper level items and it includes most of the formal constraints. *Client* describes the classes that use the components. *Component* describes the abstract interface for every component and *Component.operation* represents the methods that are called for components. *Leaf* represents class components with no substructure and *Leaf.operation* the concrete implementations for methods described by *Component.operation*. *Composite* represents the components that have child components and *Composite.operation* describes the concrete implementations for *Component.operation*. The work items, their templates, and their constraints are specified in more detail in the following.

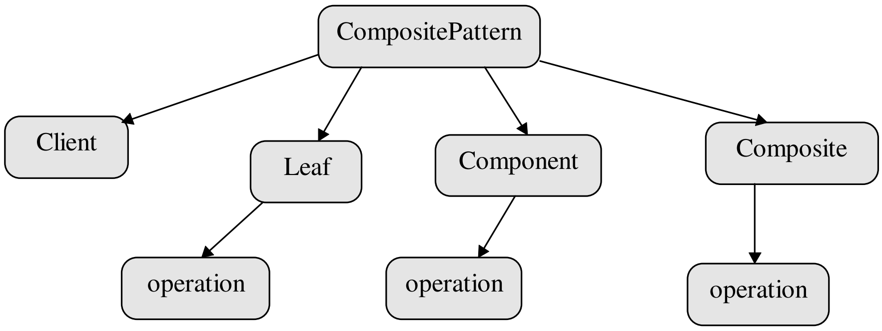

Figure A.1: The work items of the *Composite* pattern

```
//==============================================================================================
itemName = "CompositePattern"
itemType = "pattern"
atLeastCount = 1
atMostCount = 1
roleDefinitions = {}
subItems = {Client, Leaf, Component, Composite}

// Constraints:

// Every Leaf implements the Component.
∀x ∈ Leaf.Leaf, ∃y ∈ Component.Component:
  implementsInterface(x, y)

// Every Composite implements the Component.
∀x ∈ Composite.Composite, ∃y ∈ Component.Component:
  implementsInterface(x, y)

// The method Composite.operation.operationName calls Component.operation.
∀x ∈ Composite.operation.operationName, ∃y ∈ Component.operation.operationName:
  callsMethod(x, y)

//==============================================================================================
itemName = "Component"
itemType = "interface"
atLeastCount = 1
atMostCount = 1
roleDefinitions = {Component}
subItems = {operation}

// Template:

interface <def>Component</def> {
  <expansion> Component.operation </expansion>
}

// Constraints:

// There exists exactly one Component (in this pattern instance):
∃!x ∈ Component.Component[24]

//==============================================================================================
itemName = "Component.operation"
itemType = "method"
atLeastCount = 1
atMostCount = N
roleDefinitions = {returnType, operationName, arguments}
subItems = {}
```

[24] We use ∃! to denote "there exists exactly one".

```
// Template:

  <def>returnType</def> <def>operationName</def>(<def>arguments</def>);

//==========================================================================================
itemName = "Leaf"
itemType = "class"
atLeastCount = 1
atMostCount = N
roleDefinitions = {Leaf}
subItems = {operation}

// Template:

public class <def>Leaf</def> implements <ref>Component.Component</ref> {
  <expansion> Leaf.operation </expansion>
}

//==========================================================================================
itemName = "Leaf.operation"
itemType = "method"
atLeastCount = 1
atMostCount = N
roleDefinitions = {}
subItems = {}

// Template:

  <ref>Component.operation.returnType</ref>
  <ref>Component.operation.operationName</ref>(<ref>Component.operation.arguments</ref>);

//==========================================================================================
itemName = "Composite"
itemType = "class"
atLeastCount = 1
atMostCount = N
roleDefinitions = {Composite, children}
subItems = {operation}

// Template:

public class <def>Composite</def> implements <ref>Component.Component</ref> {
  protected Vector <def>children</def>;
  <expansion> Composite.operation </expansion>
}

// Constraints:

// The type of the field Composite.children is "Vector".
∀x ∊ Composite.children:
  isTypeOf(x, "Vector")

//==========================================================================================
itemName = "Composite.operation"
itemType = "method"
atLeastCount = 1
atMostCount = N
roleDefinitions = {returnType, operationName, arguments}
subItems = {}

// Template:

  <def>returnType</def> <def>operationName</def>(<def>arguments</def>) {
    for (Enumeration e = <ref>Composite.children</ref>.elements(); e.hasMoreElements(); ) {
      ((Component) e.nextElement()).<ref>Component.operation.operationName</ref>
        (<ref>Component.operation.arguments</ref>);
    }
  }

//==========================================================================================
itemName = "Client"
itemType = "class"
atLeastCount = 1
atMostCount = N
roleDefinitions = {Client}
subItems = {}

// Template:
```

```
public class <def>Client</def> {
}
```

Abstract Factory

The *Abstract Factory* consists of the following work items (figure A.2). *AbstractFactoryPattern* combines the upper level items and it includes most of the formal constraints. *Client* represents the class that uses the abstract factory. *AbstractProduct* describes the class for products that clients use. *AbstractFactory* is an interface for creating products. *AbstractFactory.createProduct* represents the creation methods. *ConcreteFactory* describes the classes that actually create products and item *ConcreteFactory.createProduct* represents the creation methods. *ConcreteProduct* describes the concrete products that are created. The work items, their templates, and their constraints are specified in more detail in the following.

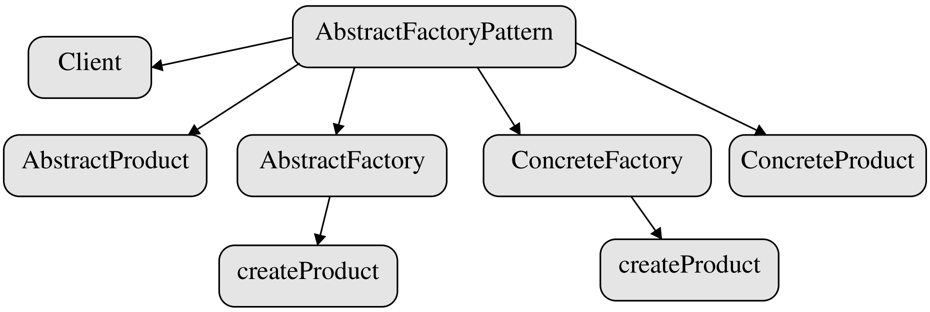

Figure A.2: The work items of the *Abstract Factory* pattern

```
//=============================================================================================
itemName = "AstractFactoryPattern"
itemType = "pattern"
atLeastCount = 1
atMostCount = 1
roleDefinitions = {}
subItems = {Client, AbstractProduct, AbstractFactory, ConcreteFactory, ConcreteProduct}

// Constraints:

// Every AbstractFactory declares methods that return AbstractProducts.
∀x ∊ AbstractFactory.AbstractFactory, ∀y ∊ AbstractProduct.AbstractProduct:
  declaresMethodWithReturnType(x, y)

// Every ConcreteFactory implements an AbstractFactory.
∀x ∊ ConcereteFactory.ConcreteFactory, ∃y ∊ AbstractFactory.AbstractFactory:
  implementsInterface(x, y)

// Every createProduct method of each ConcreteFactory instantiates some ConcreteProduct
// and returns it as an AbstractProduct.
∀x ∊ ConcreteFactory.createProduct, ∃y ∊ ConcreteProduct.ConcreteProduct:
  instantiatesClass(x, y) AND hasReturnType(x, y)

// Every ConcreteProduct implements an AbstractProduct and every ConcreteProduct
// is instantiated in some createProduct method of each ConcreteFactory.
// The instantiated ConcreteProducts are returned as AbstractProducts.
∀p ∊ ConcreteProduct.ConcreteProduct, ∃q ∊ AbstractProduct.AbstractProduct:
  implementsInterface(p, q) AND (
    ∀y ∊ ConcreteFactory.ConcreteFactory, ∃z ∊ y.createProduct:
      instantiatesClass(z, p) AND hasReturnType(z, q)
  )

// Clients must not call the methods of any ConcreteFactory or ConcreteProduct.
```

```
∀x ∈ allMethods(Client.Client), ∀y ∈ ConcreteFactory.ConcreteFactory, ∀z ∈
       ConcreteProduct.ConcreteProduct:
  NOT (callsMethod(x, y) OR callsMethod(x, z))

// Clients must not use the attributes of any ConcreteFactory or ConcreteProduct.
∀x ∈ allMethods(Client.Client), ∀y ∈ ConcreteFactory.ConcreteFactory, ∀z ∈
       ConcreteProduct.ConcreteProduct:
  NOT (refersField(x, y) OR refersField(x, z))

//==============================================================================================
itemName = "AbstractFactory"
itemType = "interface"
atLeastCount = 1
atMostCount = 1
roleDefinitions = {AbstractFactory}
subItems = {createProduct}

// Template:

interface <def>AbstractFactory</def> {
  <expansion> AbstractFactory.createProduct </expansion>
}

// Constraints:

// There exists exactly one AbstractFactory (in this pattern instance).
∃!x ∈ AbstractFactory.AbstractFactory

//==============================================================================================
itemName = "AbstractFactory.createProduct"
itemType = "method"
atLeastCount = 1
atMostCount = N
roleDefinitions = {createProduct, arguments}
subItems = {}

// Template:

  <ref>AbstractProduct.AbstractProduct</ref> <def>createProduct</def>(<def>arguments</def>);

//==============================================================================================
itemName = "ConcreteFactory"
itemType = "class"
atLeastCount = 1
atMostCount = N
roleDefinitions = {ConcreteFactory}
subItems = {createProduct}

// Template:

public class <def>ConcreteFactory</def>
  implements <ref>AbstractFactory.AbstractFactory</ref> {
  <expansion> ConcreteProduct.createProduct </expansion>
}

//==============================================================================================
itemName = "ConcreteFactory.createProduct"
itemType = "method"
atLeastCount = 1
atMostCount = N
roleDefinitions = {}
subItems = {}

// Template:

  <ref>AbstractProduct.AbstractProduct</ref>
       <def>AbstractFactory.createProduct.createProduct</def>(<ref>
       AbstractFactory.createProduct.arguments</ref>) {
     return new <ref>ConcreteProduct.ConcreteProduct</ref>();
  }

//==============================================================================================
itemName = "AbstractProduct"
itemType = "interface"
atLeastCount = 1
atMostCount = N
roleDefinitions = {AbstractProduct}
subItems = {}
```

```
// Template:

interface <def>AbstractProduct</def> {
}

//==========================================================================================
itemName = "ConcreteProduct"
itemType = "class"
atLeastCount = 1
atMostCount = N
roleDefinitions = {ConcreteProduct}
subItems = {}

// Template:

public class <def>ConcreteProduct</def>
  implements <ref>AbstractProduct.AbstractProduct</ref> {
}

//==========================================================================================
itemName = "Client"
itemType = "class"
atLeastCount = 1
atMostCount = N
roleDefinitions = {Client}
subItems = {}

// Template:

public class <def>Client</def> {
}
```

Prototype

The work items for *Prototype* are represented in figure A.3. *PrototypePattern* combines the upper level items and it includes most of the formal constraints. *Client* represents the classes that use the prototypes. *Prototype* is an interface for clonable prototypes and *Prototype.clone* is the abstract method for cloning. *ConcretePrototype* describes the concrete classes that are used as prototypes and *ConcretePrototype.clone* is the concrete clone method. The work items, their templates, and their constraints are specified in more detail in the following.

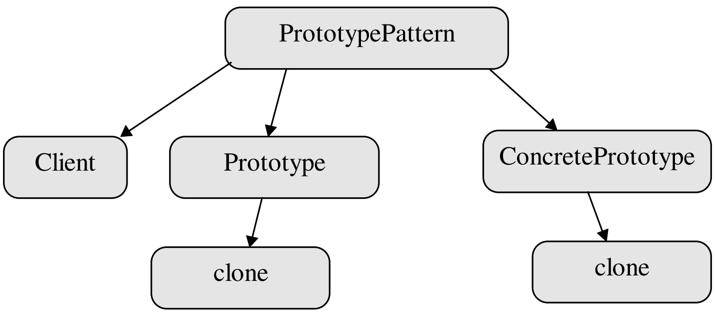

Figure A.3: The work items of the *Prototype* pattern

```
//==========================================================================================
itemName = "PrototypePattern"
itemType = "pattern"
atLeastCount = 1
atMostCount = 1
roleDefinitions = {}
subItems = {Client, Prototype, ConcretePrototype}

// Constraints:
```

```
// Every clone method instantiates some ConcretePrototype and has the interface Prototype as
// its return type:
∀x ∈ Prototype.clone, ∃y ∈ ConcretePrototype.ConcretePrototype, ∃z ∈ Prototype.Prototype:
  instantiates(x, y) and hasReturnType(x, z)

// Every ConcretePrototype implements some Prototype:
∀x ∈ ConcretePrototype.ConcretePrototype, ∃y ∈ Prototype.Prototype:
  implementsInterface(x, y)

// Clients must not call the methods of any ConcretePrototype.
∀x ∈ allMethods(Client.Client), ∀y ∈ ConcretePrototype.ConcretePrototype:
  NOT (callsMethod(x, y))

// Clients must not use the attributes of any ConcretePrototype.
∀x ∈ allMethods(Client.Client), ∀y ∈ ConcretePrototype.ConcretePrototype:
  NOT (refersField(x, y))

//==============================================================================================
itemName = "Prototype"
itemType = "interface"
atLeastCount = 1
atMostCount = 1
roleDefinitions = {Prototype}
subItems = {clone}

// Template:

interface <def>Prototype</def> {
  <expansion> Prototype.clone </expansion>
}

// Constraints:

// There exists exactly one Prototype (in this pattern instance):
∃!x ∈ Prototype.Prototype

//==============================================================================================
itemName = "Prototype.clone"
itemType = "method"
atLeastCount = 1
atMostCount = 1
roleDefinitions = {clone, arguments}
subItems = {}

// Template:

  <ref>Prototype.Prototype</ref> <def>clone</def>(<def>arguments</def>);

//==============================================================================================
itemName = "ConcretePrototype"
itemType = "class"
atLeastCount = 1
atMostCount = N
roleDefinitions = {ConcretePrototype}
subItems = {clone}

// Template:

public class <def>ConcretePrototype</def> implements <ref>Prototype.Prototype</ref> {
  <expansion> ConcretePrototype.clone </expansion>
}

//==============================================================================================
itemName = "ConretePrototype.clone"
itemType = "method"
atLeastCount = 1
atMostCount = 1
roleDefinitions = {}
subItems = {}

// Template:

  <ref>Prototype.Prototype</ref>
       <ref>Prototype.clone.clone</ref>(<ref>Prototype.clone.arguments</ref>) {
    return new <ref>ConcretePrototype.ConcretePrototype</ref>();
  }
```

```
//============================================================================================
itemName = "Client"
itemType = "class"
atLeastCount = 1
atMostCount = N
roleDefinitions = {Client}
subItems = {}

// Template:

public class <def>Client</def> {
}
```

Appendix B: Tag Language for Templates

The source code that is edited with *Template Editor* contains tags that are invisible to the user but are used by the editor. The editor makes use of these special tags to guide the specializer in the specializing process. The tags define several aspects of the source code, for example available editing functionality, specific specialization information, and tools that are available to the specializer. The specializer cannot add or remove tags, instead she must obey the rules and constraint set by these tags that are defined by the framework creator in the template files.

Before going further with the accurate tag syntax, two useful terminal symbols are defined:

- `<id>` – Describes an identifier that follows the Java lexical definition for identifiers.
- `<string>` – Describes a double quote delimited string that follows the Java lexical definition for strings. In addition, the string cannot contain the character combination '*/'.

In terms of these terminals, the following non-terminals can be introduced:

- `<tag_name>` – Corresponds to the type name of the tag and simply equals to an identifier.
- `<path>` – Describes a list of identifiers separated by dots, called path. This structure can be used to describe hierarchically ordered identifiers, such as Java package/class-hierarchy.
- `<tool_list>` – Describes a list of tools, i.e., set of subclasses to the class named Tool. Each tool is a structure of Java-class name with its corresponding package name. Each definition of a tool also has a (possibly empty) set of parameters, with named association.

```
<tag_name> ::= <id>
<path> ::= [ <id_list> . ] <id>
<id_list> ::= <id_list> . <id> | <id>
<tool_list> ::= <tool_list> , <tool> | <tool>
<tool> ::= <path> ( [ <param_list> ] )
<param_list> ::= <param_list> , <param> | <param>
<param> ::= <param_name> = <param_value>
<param_name> ::= <id>
<param_value> ::= <string> | <path>
```

After defining the preceding non-terminals, we will return back to the syntax of tags. The tags themselves correspond to Java comments '/*...*/'. They are a subset of Java comments, standing out from other comments by an exclamation mark right after the '/*'-symbol starting the comment.

```
<tag> ::= /*! <content> */
```

The exclamation mark is followed by the actual content of the tag, which defines the meaning of the tag. Tags are categorized into following three categories: (1) single tags, (2) begin tags, and (3) end tags

```
<content> ::= <single_tag> | <begin_tag> | <end_tag>
```

At the simplest, a tag is an individual mark independent of other tags in the source code. In addition, tags can be used to delimit a portion of source code and set certain attributes for a region. The begin- and end-tags are used for this purpose. For each begin-tag there must exist a corresponding end-tag. The enclosed region is in the scope of the two tags in question and is called a *field*.

A single tag always begins with a type name. This name is an identifier which describes what kind of functionality the tag provides. This name may be followed by an arbitrary set of attributes that specify the meaning of the tag. The attributes are parameters to the tag and are associated by a name.

```
<single_tag> ::= <tag_name> [ <attr_list> ]
```

A begin-tag starts with a keyword `BEGIN`. This keyword is followed by the type name and list of attributes, analogous to the single tag.

```
<begin_tag> ::= BEGIN <tag_name> [ <attr_list> ]
```

An end-tag starts with keyword `END`, optionally followed by the type name of the corresponding begin-tag. The type name is optional, because the tag structure is hierarchical and the corresponding begin-tag may be determined also without the type name as well.

```
<end_tag> ::= END [ <tag_name> ]
```

Two types of tags are now introduced; `HOTSPOT` and `LINK`. These types are described later in detail with the corresponding attributes. The attributes, as mentioned, are lexically bound attributes to the tags, which specify the meaning of the tags. The list of attributes is described in the syntax with a non-terminal `<attr_list>`. This describes a list of an arbitrary number of attributes separated by semicolons.

```
<attr_list> ::= <attr_list> ; <attr> | <attr>
```

Each attribute specification begins with the attribute's name. This is optionally followed by the value of the attribute. The value may be a string, a path (including a simple identifier), or a list of tools. Attributes without a value are also useful, as they may be used as boolean indicators. The values are interpreted in the context of the tag and the attribute in question.

```
<attr> ::= <attr_name> [ = <attr_value> ]
<attr_value> ::= <string> | <path> | <tool>
```

Common attributes

Attributes common to all tags are called `ID` and `TITLE`.

ID

An `ID`-attribute gives a name to a specific tag. The value of the `ID`-attribute must be an identifier. This identifier is a name that identifies the tag from other tags. It must be unique to other identifiers in the same document or template. The identifiers are used in cross-references between tags. Tags without valid names cannot be referenced.

The attribute is of the form

```
ID = <id>
```

TITLE

In general, the tags are invisible to the specializer. This is the case unless the `TITLE`-attribute of a tag is set. With this attribute, it is possible to make the tag and the corresponding end-tag (if any) visible. If the title attribute has a string value, the string is shown to the specializer in the place of the tag. Optionally the string value may be omitted, in which case a special icon is displayed to indicate the tag.

The attribute is of the form

```
TITLE [ = <string> ]
```

HOTSPOT

A hot spot is a special field delimited by a begin- and end-tag. The semantics of a hot spot is defined in the corresponding begin tag. Hot spot tags define the behavior of the hot spots. The features include disabling/enabling editing for the enclosed region, linking with other hot spots and support for special editing tools. The type name for hot spot tags is `HOTSPOT`. The name is followed by a list of attributes, which include (in addition to common attributes) `EDIT`, `INCLUDE`, `IN`, `TOOL`, `OUT`, and `REFERENCE`. Each attribute may occur at most once in the tag.

The begin-tag of hot spot is of the form:

```
BEGIN HOTSPOT [ <attr_list> ]
```

Within the hot spot, certain properties uphold. The inner hot spot overrides the properties of the outer one, whenever a hot spot occurs inside another one. Therefore it is possible to define a whole document as a one big hot spot, thus defining the default behavior of the document. However, by inserting hot spots within the document, the behavior of selected parts of the document can be altered — from the viewpoint of the specializer.

EDIT

The `EDIT`-attribute, when occurring in the hot spot, makes it possible for the specializer to edit the hot spot contents. If there does not exist any hot spot with the `EDIT`-attribute in a document, the specializer is unable to edit it. Only hot spots that include this attribute can be edited by the specializer. The regions where editing is allowed are shown to the specializer on different background color in contrast to read-only parts of the source code.

The attribute is of the form

```
EDIT
```

INCLUDE

With `INCLUDE`-attribute, the hot spot field may be filled with content of another hot spot, referenced with the path value of this attribute. The path consists of the document name and the hot spot name within that document.

The attribute is of the form

```
INCLUDE = <path>
```

On implementation level, a global instance of TemplateRepository class interprets the path and returns the string that must be included in the referencing hot spot. When also the `EDIT`-attribute is set, the target of the `INCLUDE`-attribute is copied from its original location into the hot spot. At the same time the `INCLUDE`-attribute is removed from the hot spot to avoid recopying the contents. The possible changes made to the hot spot are therefore never overwritten. On the other hand, when the `EDIT`-attribute is *not* set, the included field is not copied. Instead the changes made to the referenced hot spot are always reflected in the referencing hot spot.

REFERENCE

The REFERENCE-attribute functions similarly to the `INCLUDE`-attribute when the `EDIT`-attribute is not set. However, the contents of the referenced hot spot is not displayed within the hot spot; instead a hyperlink to the referenced hot spot is generated. The title of the hyperlink is defined by the `TITLE`-attribute of the same hot spot tag. If the title is not set, a special symbol is displayed to show the hyperlink.

The attribute is of the form

```
REFERENCE = <hotspot>
```

IN

Whenever the specializer moves the caret into the hot spot, or clicks the hot spot with the mouse, a set of tools may be launched. Recall that tools are Java classes of type Tool. A useful exemplar tool includes information viewer, which displays hot spot sensitive help text in a separate pane.

The tools are presented in a special `IN`-attribute, which takes a list of tools (`<tool_list>`) as a value.[25] The attribute is therefore of the form

```
IN = <tool_list>
```

For each tool name contained in the list, an instance of the corresponding class is created and launched with its `execute()`-method. The tool receives a set of lexically bound parameters as defined in `<tool_list>`. The tool must clean up when its `terminate()`-method is called. This method is called when the hot spot is exited later on and the launched tool is no longer needed.

[25] Given an attribute `IN=InfoViewer(info="information")`, a tool called *InfoViewer* is executed whenever the specializer enters the hot spot. It has a parameter called `info`, associated with a string `information`. It is supposed that this tool prints out the string given as parameter to a separate info field provided by *Template Editor*.

TOOL

The creator of the framework may provide special tools that may be used to edit the contents of the hot spot. The tools may provide an optional view or alternate editing mechanism for the text contained in the hot spot. The tool may be the only way for the specializer to alter the contents of the hot spot, or it may be an optional method. Either way, this kind of tools — which are launched explicitly from the toolbar by the specializer (as opposed to the tools declared with the IN-attribute) — are provided in the TOOL-attribute of the hot spot.

The attribute is of the form

```
TOOL = <tool_list>
```

If the EDIT-attribute is set in the same tag, the tools are optional to the specializer; the specializer may edit the field also traditionally without the tools. In the opposite case, the specified tools provide the only way to alter the hot spot contents.

The tools available are provided to the specializer in a separate hot spot-sensitive toolbar that is included in the *Template Editor* user interface.

OUT

Just like the IN-attribute declares tools that are executed whenever the specializer enters a hot spot with the caret (or mouse click), the `OUT`-attribute declares the operations that are carried on when the specializer exists the hot spot.[26] One of the implicit operations that need not to be mentioned in the `OUT`-attribute, is termination of all tools launched with the `IN`-attribute. This is done, as mentioned before, by a call to `terminate()` for each launched tool.

The attribute is of the form

```
OUT = <tool_list>
```

Typically, the `OUT`-attribute contains tools which check that the source code contained in the hot spot is valid in some sense, make the necessary changes or start macros that update the associated fields based on the edited hot spot contents.

LINK

`LINK` is a type name of single tags that are used to create hyperlinks right within the Java source code. The target of such hyperlink is any named tag (named with an `ID`-attribute) contained in some of the documents accessible

[26] Suppose *InfoViewer* was started with the `IN`-attribute of some hot spot. When the hot spot is exited (the specializer moves the caret out of the hot spot) all the tools launched previously are terminated. *InfoViewer* did not begin a new thread however, like it could have done. There is still something to be done with the *InfoViewer* tool at this point. The state of info field before launching *InfoViewer* must be restored. Therefore, this must be implemented in the `terminate()`-method of *InfoViewer*.

by the editor. The target of a `LINK`-tag is determined with the `TARGET`-attribute, whereas the `TITLE`-attribute gives the hyperlink a name that is displayed to the specializer with underlined text. Without the `TITLE`-attribute, the link tag is invisible to the specializer. Whenever the `TITLE`-attribute is present, but the value of that attribute is missing, the link is viewed as a special symbol.

The tag is of the form

```
LINK [ <attr_list> ]
```

TARGET

The `TARGET`-attribute presents the target of the link. This is described with a path string which contains the path of the document and the name of the target tag in that document. If the path begins with an empty element (i.e., it begins with a dot) the target is searched from the same document. When the `TARGET`-attribute is missing, the `LINK`-tag does not function as a link, but a passive title that does not respond to any user interaction.

The attribute is of the form

```
TARGET = <path>
```

Appendix C: Alphabetical Listing of FRED's Classes

In the following we have included an alphabetical listing of the classes in figure 5.1. However, for simplicity only the most important methods and attributes have been described.

Application

`Application` is a `Project` that is created for application construction.

Associations:

- `frameworks` — The frameworks chosen for this application.

Operations:

- Operations related to managing the associations and attributes[27].

ApplicationComposer

`ApplicationComposer` (see also chapter 3.3) is used for selecting frameworks to be specialized and for selecting the appropriate work lists for the current situation. After that the work list is executed to build a working application. Note that many frameworks can be used together to form an application.

Associations:

- `application` — The application that is being constructed.
- `applicationWizard` — Executes the selected work list one item at a time under the user's guidance.
- `frameworkBrowser` — Used for selecting frameworks and their work lists for making an application.

ApplicationWizard

`ApplicationWizard` is a user interface component that shows the current work list and lets the user select items to be done. Each item is executed using one of the tools that are specified in the `WorkItem` object. The user selects the tool if there are more than one tools available.

Associations:

- `activeItem` — The item that is being done by the wizard.

Category

`Category` describes a group of related design patterns.

[27] These operations are not mentioned in any of the classes below, although in the actual implementation they are of course necessary.

Associations:

- `patterns` — The patterns included in this category.

ClassBrowser

`ClassBrowser` provides a graphical tree representation of a code hierarchy related to a project (an application or a framework).

ClassDiagram

`ClassDiagram` is a user interface component for graphical chart representation of an application or a framework.

ClassFactory

`ClassFactory` is a user interface component that lets the user select one or more base classes for a new subclass to be created. The subclass will then be created according to a suitable work item and its template.

CodeElement

`CodeElement` is an internal representation of a code entity (e.g., a Java class, a method, or an attribute).

Constraint

`Constraint` describes a post condition that has to hold for a work item. It is implemented as a predicate clause that includes references to roles defined in the item the constraint is associated with (or in one of its subitems). When checking a constraint for a specific `WorkItemInstance` the constraint specification is obtained via the `WorkItem` and then the roles are bound as defined in `RoleBinding` objects of the `WorkItemInstance`. The actual constraint check is implemented according to the *Strategy* design pattern.

Associations:

- `refersTo` — The roles that this constraint refers to.

Attributes:

- `constraintClause` — An object that corresponds to the user given textual predicate clause (i.e. an object that implements the `check()` method with similar semantics).

Operations:

- `boolean check(roleBindings)` — Executes the predicate using the given role bindings.

CompositionTool

`CompositionTool` is a user interface component that lets the user graphically create and initialize objects and compose various object structures.

DesignPattern

A formal description of a design pattern. In the FRED toolset every design pattern specification consists of work lists (one for each pattern variation), a set of HTML documentation files, and category information.

Associations:

- `categories` — The weighted associations to pattern categories.
- `documents` — The (HTML) document files.
- `workLists` — The descriptions for different pattern variations.

ElementHierarchy

`ElementHierarchy` represents the hierarchy of code elements (e.g., class hierarchy) in the current project.

ExternalFile

`ExternalFile` represents a descriptor for a file related to a FRED project (e.g., a template file, an HTML-document, a source code file, or a class file).

ExpansionTool

`ExpansionTool` takes care of including subtemplates into their parent templates as needed. For more details see chapter 5.2.4.

Framework

`Framework` represents a `Project` that is created for framework construction.

FrameworkBrowser

`FrameworkBrowser` is a user interface element for browsing framework descriptions and for selecting an appropriate framework and its work list for constructing the desired application.

Associations:

- `frameworks` — The available frameworks.

FrameworkComposer

`FrameworkComposer` is the user interface component for the framework construction part of the FRED toolset (see also chapter 3.2). It is used to create new frameworks by associating existing code and documentation to the framework, by instantiating design patterns, and by building new work lists for intended framework users.

Associations:

- `patternSelectionWizard` — Used for selecting suitable design patterns to be instantiated.
- `workListEditor` — Used for creating work lists for the framework.

FRED

`FRED` is the main class of the whole FRED toolset. It consists of four main components `PatternComposer`, `FrameworkComposer`, `ApplicationComposer`, and `Visualization`. The FRED class manages the toolbar from which any of these components can be launched. It holds also a reference to the currently active project. There can be only one active project at a time.

Associations:

- `project` — The active project.

Operations:

- Project handling operations.

PatternComposer

`PatternComposer` is the user interface component for the pattern composition part of the FRED toolset (see also chapter 3.1). PatternComposer is used to build new patterns, i.e. creating work lists, associating HTML documents, and categorizing the patterns.

Associations:

- `categories` — Existing categories.
- `designPatterns` — Existing design patterns.
- `workListEditor` — `PatternComposer` uses the `WorkListEditor` to create work lists for pattern variations.

PatternSelectionWizard

`PatternSelectionWizard` is a user interface component that helps the user to select an appropriate pattern for the problem at hand.

Associations:

- `categories` — The pattern categories.
- `patterns` — The available design patterns.

Project

`Project` represents an active framework construction or specialization session.

Associations:

- `activeWorkList` — The active work list.
- `elementHierachy` — The hierarchy of `CodeElements` in the current project.
- `files` — The external files (documents, source codes, binaries, etc.) of the project.
- `workListInstances` — The work lists under execution. Each work list corresponds to a framework that is being applied.

Attributes:

- `name` — The name of the project.

Role

`Role` describes a name located in code (text), and thus indirectly a structure with certain semantics. A role can refer to, for example, a class name in a `TemplateFile`. Roles are also used when specifying `Constraints` for certain semantic structures. Each role must be defined in exactly one `WorkItem`. We talk about binding a role when a certain concrete name is associated with the role. This can be compared to an assignment to a variable or a macro. Note that role bindings are searched via `WorkItemInstance` by indexing with the role name. This is like a fetch from a symbol table.

Associations:

- `usedInConstraints` — The constraints (`Constraint`) that refer to this role.
- `usedInTemplates` — The template files (`TemplateFile`) that refer to this role.

Attributes:

- `name` — The name of the role.

RoleBinding

`RoleBinding` describes the concrete binding between a name in a template file and a role.

Associations:

- `itemInstance` — The owner of this role binding.

- `observers` — Descriptors of template files that must be updated when the value of this role (the name that it is bound to) changes.
- `role` — The role that is bound.

Attributes:

- `name` — The string that this role is bound to. The string is expanded into template files that refer to this role.

RoleTool

`RoleTool` takes care of the role binding mechanism of the FRED tool. It updates role bindings and broadcasts changes to every observer of the updated role (i.e. to the templates that refer to the role). For more details see chapter 5.2.4.

ScenarioDiagram

`ScenarioDiagram` is a user interface component for graphical symbolic execution of an application or a framework.

TemplateEditor

`TemplateEditor` is a user interface component that guides the user as she edits text files (e.g., source code). The files include special tags that are embedded in Java comments. Tags are used, for instance, to control which areas of text are editable, to propagate changes in code to other text locations and files, to implement hypertext links and help texts, and to invoke other (higher-level) tools to produce code. For more information on the editor and the tag language, see chapter 3.3 and appendix B.

TemplateFile

`TemplateFile` is a descriptor of a text file that acts as a template for generated code (text). `TemplateFiles` are written in the tag language described in appendix B. Since tag language constructs are embedded in Java comments, the template files can be used as source code files without any extra modifications.

Associations:

- `assignments` — The roles for which the user can assign new values through this template.
- `filesToBeExpanded` — The other templates to be expanded into this template (using `ExpansionTool`).
- `usages` — The roles that this template refers to.

TemplateMaker

`TemplateMaker` is an editor that is used for creating, editing, and storing `TemplateFiles`. It is similar to high-level HTML editors. `TemplateMaker` provides an easy-to-use and intuitive user interface for the features of the tag language. The user can select areas of text with the mouse and assign appropriate tag language properties for the selected text area.

Associations:

- `availableTools` — Tools that can be selected for invocation for a certain text area.
- `templateFile` — The current template file.

Tool

`Tool` is a common interface for all tools that can be used to do `WorkItems`. They are invoked, e.g., from the `TemplateEditor` or from the `ApplicationWizard`.

Operations:

- `initialize()` — Initializes the fields of the editor. E.g., it can read the source code in order to set up radio buttons, text fields, etc.
- `getText()` — Returns the source code block which is based on the current values in the editor. The generated code can then be added to the actual hot spot in the editor.
- `execute()` — Opens the tool.
- `terminate()` — Closes the tool.
- `update()` — Updates the tool controls.

Visualization

`Visualization` (see also chapter 3.4) is a common user interface component for the visualization of the static and dynamic semantics of the implementation-level entities of the current project (a framework or an application).

Associations:

- `classBrowser` — Shows static structure as a tree.
- `classDiagram` — Shows static structure as a graphic chart.
- `hierarchy` — The active code element hierarchy (usually a class hierarchy).
- `scenarioDiagram` — Runtime visualization as a scenario.

WorkItem

`WorkItem` describes a task that has to be done in order to proceed in specializing a framework (or instantiating a design pattern when creating a framework). `WorkItem` usually describes the information needed for generating[28] and validating[29] a small fragment of code (or other text). On the other hand, `WorkItem` may also be used just to group related work items together. This kind of a work item may include `Constraints` concerning its child items. A work item may have a group of `CodeElements` associated with it. These might be used by a special tool that executes the item[30].

A work item can be associated with a `TemplateFile` that represents source code-level entities, such as interfaces, (sub)classes, and their members that must be implemented when deriving an application from a certain framework. When the user implements an entity as described in the corresponding work item, the item is said to be *done* (or *executed*). Items in a work list can be done in any order unless a specific implementation order is requested. Each type of work item will have some kind of `Tool` associated with it (or many of them). The tools are used to execute the item by, for example, editing its template file. More complex work items can be formed by grouping many primitive work items together and by associating them with a special tool. By default, item templates are edited using the `TemplateEditor`.

The states of a work item or its components are not changed during the execution of the `WorkList` they belong to. Instead, we use explicit `WorkItemInstance` objects to reflect and keep track of the usage of work items. In fact, the same work item can be used in different work lists at the same time. The user has to save the item under a different name to use it as a base for a new item.

Associations:

- `codeElements` — Code elements that are utilized when executing the item.
- `constraints` — Formal constraints for the item.
- `itemRelations` — This is a short hand notation for several associations, for example, `subItems` (item name hierarchy), `connectedItems` (“synchronized items”), and `relatedItems` (documentation).
- `roleDefinitions` — The roles defined in this item. Every role is defined in exactly one `WorkItem`.
- `templateFile` — The template that describes the structure of the item.
- `tools` — The tools that can be used to do this item.

Attributes:

- `atLeastCount` — The minimum number of instances that should be created from this item (within one active work list).

---

[28] This is achieved by utilizing the possible `TemplateFile` associated with the item.

[29] Validation is done by checking the possible `Constraints` associated with the item.

[30] An example of such tool is `ClassFactory` that lets the user choose (from a set of suitable candidates) a superclass for a new subclass. In this situation attribute `javaElements` refers to the superclass candidates.

- `atMostCount` — The maximum number of instances that can be created from this item (within one active work list).
- `description` — A short description of the work item.
- `name` — The name of the item.
- `type` — The type of the item (e.g., "method", "class" or "interface").

Operations:

- `WorkItemInstance createInstance()` — Creates a new instance from the work item and initializes the instance with a `TemplateFile` and empty copies of the `RoleBindings`. In addition, the instance stores a reference to its creator.

WorkItemInstance

A `WorkItemInstance` object is created when the user executes a `WorkItem`. The number of instances created based on a work item must lie between `WorkItem.atLeastCount` and `WorkItem.atMostCount`.

Associations:

- `item` — The work item whose instance this item is.
- `itemRelations` — This is a short hand notation for several associations, for example, `subItems` (item name hierarchy), `connectedItems` ("synchronized items"), and `relatedItems` (documentation).
- `roleBindings` — The concrete role bindings for this item.
- `templateInstance` — The template file descriptor associated with this item.
- `workListInstance` — The work list to which this item belongs.

Attributes:

- `instanceNumber` — The ordinal number of the instance.

Operations:

- `associateRoles(names)` — Associates roles to the given names.
- `checkConstraints()` — Executes all constraint checks.

WorkList

A `WorkList` describes a sequence of actions (a set of `WorkItem` objects) that needs to be executed in order to create one or more applications from a `Framework` or to instantiate a design pattern. In `PatternComposer` work lists are used for formal definition of design patterns. In `FrameworkComposer` they are used for the implementation of design pattern structures, as well as for storing framework specialization instructions. *`Application Composer`*, in turn, uses work lists to assist the user when she is applying a framework.

Associations:

- `designPattern` — The possible design pattern from which this work item has been retrieved.
- `documents` — Documentation files (for example, a set of HTML pages).
- `WorkItems` — The individual items in the work list.

Attributes:

- `description` — A short description of the work list. All help texts and descriptions might be strings or text files.
- `name` — The name of the list.

Operations:

- `WorkItemInstance createInstance()` — Creates `WorkListInstance` objects.
- Operations related to saving and loading the work list and its items. We use *proxies*: the loading of items is delayed until someone really needs them.

WorkListEditor

`WorkListEditor` is used for making work lists and their items, as well as for arranging item hierarchies.

Associations:

- `availableTools` — Tools that can be selected to execute a certain item.
- `templateMaker` — Used for creating, editing, and storing template files.
- `workItem` — The current item.
- `workList` — The work list under construction.

WorkListInstance

`WorkListInstance` is a work list taken into use. It keeps track of the work item instances.

Associations:

- `items` — Work item instances of this work list sorted according to their types.
- `project` — The project to which this work list instance belongs to.
- `workList` — A work list whose instance this is.