Tool Support for Pattern Oriented Design: GUI, Database, and Pattern Browser

Hengyi Xue

Problem Report

ABSTRACT

Tool Support for Pattern Oriented Design: GUI, Database, and Pattern Browser

Hengyi Xue

This report documented the development process of a CASE tool that supports pattern oriented analysis and design (POAD) methodology. Three components of the tool: GUI, database and pattern browser were discussed. Documentation for each component includes analysis, design and implementation.

Table of Contents

1. INTRODUCTION

1.1 Background

1.1.1 What is a Design Pattern

In recent years, design patterns have attracted the interest of researchers and practitioners in the object-oriented community. Design patterns are high level design elements that provides solution to recurring problems. Gamma et. al. categorized a design pattern as a description of communicating objects and classes that are customized to solve a general design problem in a particular context [1]. A design pattern tries to capture the essential insight into a problem so that others may learn from it and make use of it in similar situations. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

The format to describe a design pattern is essential to convey the knowledge it encompass and to encourage its use in practice. In [1], Gamma et. al. suggested the use of the following template to describe a pattern.

Pattern Name and Classification

A pattern must have a meaningful name. The pattern's name conveys the essence of the pattern succinctly. Good pattern names form a vocabulary for discussing conceptual abstractions. Sometimes a pattern may have more than one commonly used or recognizable name in the literature. In this case it is common practice to document these nicknames or synonyms under the heading of Aliases or Also Known As. Some pattern forms also provide a classification of the pattern in addition to its name.

Intent

A statement of the problem which describes its intent: the goals and objectives it wants to reach within the given context and forces.

Also Known As or Aliases

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help a designer understand the more abstract description of the pattern that follows.

Applicability

The preconditions under which the problem and its solution seem to recur, and for which the solution is desirable. This tells the users the pattern's applicability. It can be thought of as the initial configuration of the system before the pattern is applied to it.

Structure

A graphical representation of the classes in the pattern.

Participants

The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

Consequences describe how the pattern supports its objectives, the trade-offs and results of using the pattern.

Implementation

The pitfalls, hints, or techniques that a designer should be aware of when implementing the pattern, as well as possible language-specific issues.

Sample Code

Code fragments that illustrate how the pattern might be implemented in certain languages.

Known Uses

One or more sample applications of the pattern which illustrate: a specific initial context; how the pattern is applied to, and transforms, that context; and the resulting context left in its wake. Examples help the reader understand the pattern's use and applicability. Visual examples and analogies can often be especially illuminating. An example may be supplemented by a sample implementation to show one way the solution might be realized. Easy-to-comprehend examples from known systems are usually preferred.

Related Patterns

Design patterns that are closely related to the one being described. The difference between them and suggestions on which one to use under what conditions.

Examples of design patterns used in this report are shown in Appendix A.

1.1.2 Visual Modeling and UML

As software industry moves to develop systems of greater and greater complexity, our ability to manage that complexity follows our ability to visualize our systems above the level of raw lines of code. Visual modeling is a way of thinking about problems using models organized around real-world ideas[2]. Models are useful for understanding problems, communicating with everyone involved with the project, modeling enterprises, preparing documentation, and designing programs and databases. Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building. Three elements are needed to successfully diagram and visualize a software system-a process, a notation, and a modeling tool. The Unified Modeling Language (UML) is a standard notational language

for visual modeling, which incorporated the techniques of a group of best methodologist. The goal of UML is to provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models. UML has become a de facto standard in the domain of object-oriented analysis and design. A visual modeling tool that supports UML can be expected to be easy to learn and easy to use for the users.

1.2 The Problem

In recent years, much activity has been concentrated on discovering and documenting design patterns. Little emphasis, however, has been placed on deploying these reusable designs to build software. Yacoub and Ammar[3] discussed the potential benefits of using patterns as design components in the aspect of software reuse. In software reuse, code reusability provides a low percentage of effort savings but it is the most popular and common approach. Reusable designs and frameworks are less frequent due to the complexity and difficulty of constructing generic designs for common engineering problems. Design patterns help leverage the reuse level to the design phase because they provide a common vocabulary of designs, provide means of understanding designs, and they are proven building blocks from which more complex applications can be built. The authors also proposed the methodology of pattern oriented design. High quality design patterns were specified as design components. The specification of a pattern was categorized as Recipe, Formal Specification, and Interface Specification. Design patterns qualified as design components were classified as constructional design patterns. A constructional design pattern is a design pattern that provides a solution that is an abstraction of a common design structure in the form of a UML class model. Constructional design patterns can be glued together at a high design level to define the application solution structure. The internal details of the pattern structure are hidden at high design levels (pattern models) and are traceable to lower design levels (class models). The notion of Pattern Interface was introduced to specify a pattern as a component and to show how patterns interact at the design level. Three design levels based on design components were discussed; namely the pattern level diagram, a pattern level diagram with interfaces and a detailed pattern diagram. This approach would produce designs that can be viewed at different level of abstraction. At high level of abstraction (pattern level diagram), the design is viewed as patterns and their dependency relationships. At lower level of abstraction (pattern with interface level diagram and detailed pattern diagram), the interfaces and internals of the patterns can be revealed to produce traditional object oriented class diagram.

Deploying patterns to develop complex systems is a tedious task that involves integration issues and iterative development. Automation of this process will eventually facilitate the analysis and design phase. However, current modeling tools do not explicitly support patterns as architecture construct. A tool support is needed for this methodology to be used in practice.

1.3 Related Tools

A few tools have been developed to help document and instantiate patterns. In [4], the authors listed the following tools and their approach and limitations.

Framework Adaptive Composition Environment (FACE)

For code generation and pattern instantiation purposes, the software development environment FACE (Framework Adaptive Composition Environment) [5] was developed to guide the process of instantiating patterns. It starts with a primal-schema, containing the abstract classes of a pattern and their association, then proceeds to a meta-schema for concrete classes, operations and associations.

Fragmentation Technique

The "fragmentation" technique [6] shows another approach for binding patterns together with one another and with other non-pattern classes. This approach does bind patterns together with other program artifacts but at a lower level of fragments. It is suspected that it will be difficult for large designs since it is difficult to trace and have cumbersome design views. The tool support for pattern-oriented architecture described later in this paper solves this problem at a higher level of abstraction.

PSiGene CASE tool

The application PSiGene CASE tool [7] is another tool for binding patterns from a predefined catalog with class models to form the final application. This method is application specific (building simulators) and it also generates the classes and methods for specific patterns from a catalog but doesn't link them together in a higher design level.

Code Generation

The research in IBM [8] has developed a tool for code generation from design patterns. The tool shows how code can be automatically generated for a pattern by supplying application specific information of a chosen pattern.

The Pattern-Lint

The Pattern-Lint tool [9] was introduced to check the compliance of a pattern implementation. A set of rules is defined for each design patterns and the implementation is checked against these rules. The tool is used for analysis of systems developed from patterns but does not implement a methodology to develop applications using patterns.

Hooks and Templates

A more global view of deploying patterns in design is proposed in [10], a meta-pattern called "HOOK&TEMPLATE" is presented to distinguish between pattern components that will be implemented by the user and those components that are already defined for the pattern class collaboration.

Framework Studio

The framework studio [11] is a commercial tool for storage of patterns. It allows searching and browsing catalogue of patterns and then inserting the class diagram of a selected pattern in UML tools such as Rational Rose. This tool is a general-purpose repository management tool and does not address any specific design methodology.

None of the currently available tools discusses how to assist the process of constructing designs by gluing together constructional design patterns. Pattern-oriented designs provide a good level of abstraction at the architecture level. The main concerns of most of the current tools are pattern instantiation, implementation, or code generation.

1.4 Proposed Solution

To support pattern oriented analysis and design, a tool is proposed that supports high-level designs using patterns as design components with interfaces, and integrates with existing tools for lower level designs in terms of class and collaboration diagrams. The top level architecture of the tool is presented as follows:
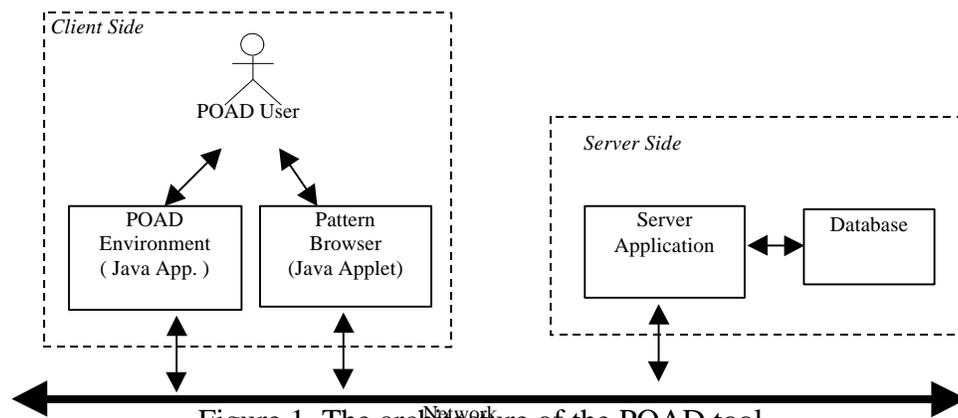
Figure 1. The architecture of the POAD tool

The tool has four components; the Pattern Oriented Analysis and Design (POAD) Environment, the Pattern Browser, the Server Application, and the Database.

Database

The database holds the repository of constructional design patterns available as reusable design components. To reuse a design pattern as a building component, the storage of the pattern in the database should facilitate the retrieval process. Each pattern stored in the repository has three constituents:

- the traditional pattern documentation, including the context, problem, forces, consequences, sample code, etc. Such template is documented in the GoF or POSA books.
- the pattern interfaces. There could be multiple interfaces for the pattern all of which are stored in the repository.
- the pattern model which holds the structure of the pattern that can be embedded into the application design. In our POAD, the design structure is represented as class diagrams.

Server Application

The server application is the interface to the database components. It authorizes connection to the database and launches the pattern browser applet to the requesting client.

Pattern Browser

This component is used to browse the database of constructional design patterns. The pattern browser is implemented as a Java Applet. The interface of the applet is composed of a browsing window that has a tree structure of patterns under pre-defined categories. The interface displays a frame for each pattern with folder structure. The tool supports the searching capability of the database.

The POAD Environment

The POAD Environment supports the POAD design methodology. The GUI supports the three views Pattern-Level view, Pattern Interfaces view, and Detailed pattern view. This component is composed of the following:

1.   *The User Interface module*. It implements the three model views Pattern-Level, Pattern Interfaces, and Detailed Pattern views.

2.   *The design engine*. This module is the application manager that creates the objects viewed by the GUI interface based on the design metamodel.

3.   *The database access module*. This module is responsible for connecting to the remote database to retrieve a pattern model file.

4.   *Local system storage and retrieval*. The design models produced by the POAD Environment user can be saved locally.

1.5 Report Structure

As part of a team to develop the tool support for Pattern Oriented Design, I have designed and implemented the GUI module of the POAD Environment, the database and the pattern browser. In this report, the analysis, design, and implementation of these modules will be discussed in detail. The report is organized into six sections. The first section introduces the background and purpose for this research project. Section 2, 3 and 4 document the development process of the GUI module, the database and the Pattern Browser separately. Section 2 and 4 are organized into three sub-sections: functional requirements, analysis and design, and implementation. Section 3 is organized into two sub-sections: E-R modeling and table definitions. Section 5 presents the conclusion for this report. References are listed in the section 6. Design patterns used in the design, user's guide for the POAD Environment and the Pattern Browser, and the source code are included in the Appendix.

2. DEVELOPMENT OF THE USER INTERFACE MODULE

2.1 Requirements

Requirement for the GUI module of the POAD Environment was taken from [4] and listed here.

1)   The tool has an easy-to-use graphical user interface

- has the same look and feel to current object oriented modeling tools such as Rational Rose

- adheres to similar notations as that of Unified Modeling Language [12]. In all diagrams, standard UML notations should be used.

2)   The GUI of the tool has:
- Three Working Areas
a) Browser: The browser maintains a list of all created diagrams.
b) Documentation: A window that displays the documentation of a selected design artifact.
c) Diagrams Window: A window that displays all pattern diagrams opened for editing.
- Standard Toolbar and Menu to facilitate accessibility to operations provided by the tool.
- Working Toolbar: The Working Toolbar has all possible artifacts and relationships that could be instantiated in a design diagram. The content of the Working Toolbar changes according to the type of the diagram selected for editing. For instance, when working at the *Pattern-Level* view we have patterns as artifacts that could be instantiated from the Working Toolbar. While working in a *Pattern Interfaces* diagram, the Working Toolbar has classes and operations as design artifacts
- Specification Window: A specification window is used for viewing and editing the specifications of a selected design artifact.

3)   The Browser Window supports three logical views:
a) *Pattern-Level* view (Section 2.1.1)
b) *Pattern Interfaces* view (Section 2.1.2)
c) *Detailed Pattern* view (Section 2.1.3)
For each view, the user shall be able to create pattern diagrams that reflect the corresponding view. For each view the Working Toolbar reflects the artifacts that could be used in developing a pattern diagram.
The following sections describe the pattern views supported by the tool.

2.1.1 Pattern-Level view

The purpose of diagrams developed under *Pattern-Level* view is to model the application as a visual composition of patterns at a high design level.

Creation: *Pattern-Level* diagrams are created under *Pattern-Level* view in the browser window.

Working Toolbar Artifacts: When editing a *Pattern-Level* diagram, the Working Toolbar shall contain the following artifacts: Notes, Note Connection, Subsystem, Pattern, Dependency Relationship, and an Interface Connector
Schematic Diagram: A *Pattern-Level* diagram can display the following artifacts. (For each artifact, the designer shall be able to use a specification window to view/edit the artifact attributes. The content of the specification window differs according to the selected artifact).

a) Patterns
Patterns are represented as rectangles with Pattern Instance Name and Pattern Type:

Pattern Instance Name: Pattern Type
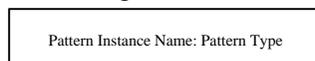
Type: each pattern instance has a type. We refer to the Type here as the well known and documented name of the pattern, for example *Observer*, *Factory*, *Strategy* patterns [1]

Instance Name: This is the name of the instance of the pattern created in the application design. You can instantiate several patterns of the same type in one design; the name of the instance will distinguish each. Example: *SensorObserver* and *UserInterfaceObserver* are instances of type *Observer.*

The specification window for a pattern displays the Name of the pattern, its type (selection from a list of pattern library: Observer, Strategy, … etc.), and the associated documentation.

b) Subsystem

A subsystem is a unit of high-level structure decomposition. It contains patterns and probably other subsystems. It has its own name space and set of owned elements. A subsystem is represented using two rectangles as shown with the subsystem Name in the upper rectangle.
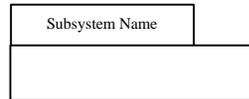
c) Dependency Relationship
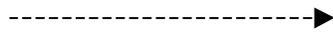
A dependency relationship is modeled by dotted line directed to the entity that the source depends on. Currently, dependency is defined as one type of relationships between patterns. A dependency indicates a semantic relationship between two pattern or two subsystems. This relationship is further refined at later design phases by translating it to an association between classes of two communicating patterns or subsystems.

The specification window of a dependency relationship displays the dependency name, role of dependee, role of dependent, and the associated documentation.

d) Notes

Notes can be attached to patterns, subsystems, or dependencies using the Note Connector. Notes have boundaries like folded paper.

e) Interface Connectors

Interface Connectors are used to connect several design sheets together. They are represented by the lollipop symbol (similar to Rose interfaces). Interface connectors have composite names; i.e., Diagram Name: Connector Name. The diagram name shows the names of the other diagrams that this current sheet connects to. Interface Connectors can be connected to subsystems or patterns. The specification window of the Interface Connector shows a list of all other design diagrams that use/provide this interface.

Figure 2 shows a sample *Pattern-Level* diagram.
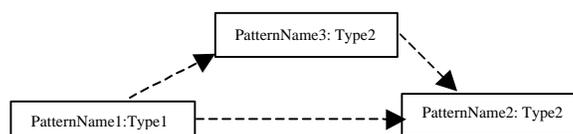
Example: The following figure shows a sample *Pattern-Level* diagram for a feedback control system developed in [13]
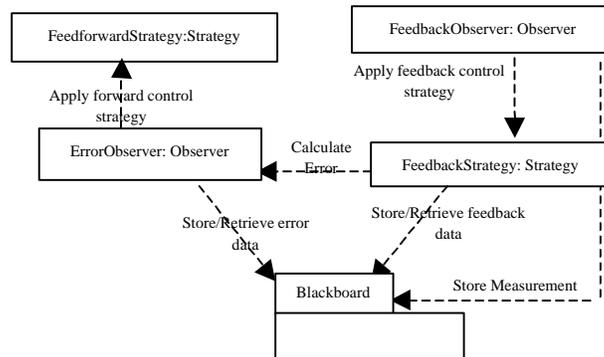
Figure 3. A *Pattern-Level* diagram for Feedback Control Application

Model Checker Rules: The following is a set of rules that the tool checks for consistency:
1. Interface Connectors can only be connected to subsystems and patterns.
2. Within one *Pattern-Level* diagram, no two Interface Connectors have the same name.
3. Subsystem and Pattern Instance names are unique
4. Dependencies are only used between Patterns, subsystems and Interface connectors.

2.1.2 Pattern Interfaces view

The purpose of diagrams developed under this view is to display the interfaces between two patterns. These two patterns are dependent on each other as specified by the dependency relationship in the *Pattern-Level* view.

Creation: For each dependency relationship between subsystems or patterns in a *Pattern-Level* diagram, we create a *Pattern Interfaces* diagram to further analyze the dependency relationship.

Working Toolbar Artifacts: When editing a *Pattern Interfaces* diagram, the Working Toolbar shall contain the following artifacts: Notes, Note Connection, and Associations
Schematic Diagram: A *Pattern Interfaces* diagram contains the following artifacts:

a) Pattern Interfaces
We should be able to represent interfaces as:
1. Classes
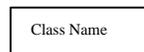
Represented as rectangles labeled with the class name adhering to the pattern frame. Classes at the interface reflect some of the internal classes of the pattern. The interface class can be either a Client or Server interface. The direction of the arrow in the association relationship determines its role.
2. Operations

11
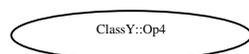
Represented as rounded ellipses labeled with the "Class Name::Operation Name". Interface operations are accessed by other design artifacts, or access other operations in other design components. The direction of the arrow (outgoing or incoming) from/to the operation specifies its role in the interaction (Required/Provided). Each operation is identified by the class name to which it belongs. This class name is one of the internal classes of the pattern.

b) Notes: Same as *Pattern-Level* diagram

c) Associations
We define three semantic types of associations:
1. Class/Class: This is the traditional class associations that can be: Aggregation, Dependency, Generalization or Composition
2. Class/Operation: An interface class can interact with an interface operation in another pattern. This type of relationship is expressed when it is still ambiguous to the designer which operation in the interface class interacts with this interface operation.
3. Operation/Operation: This is usually used at a lower design level, it shows which interface operations are interacting. The direction of the arrow would indicate the Required/Provided role of the interface. We will use a solid arrow for all possible types of associations.
The following diagram shows a *Pattern Interfaces* diagram between two patterns Pattern1 and Pattern2
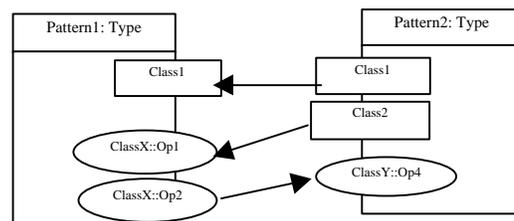
Figure 4. Pattern interface diagram

Example: Figure 4 shows the *Pattern Interfaces* diagram for the "*Calculate Error*" dependency relationship between the *FeedbackStrategy* pattern and the *ErrorObserver* pattern of figure 2.
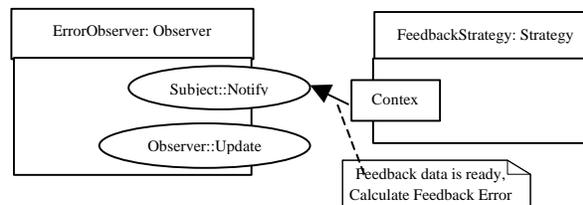
Figure 5. Pattern interface diagram for a dependency relationship in the Feedback Control Application

Model Checker Rules: The dependency relationship at *Pattern-Level* view should be completely covered at *Pattern Interfaces view*; i.e., the tool gives a model check error if a dependency relationship in a *Pattern-Level* diagram is not further decomposed associations in the *Pattern Interfaces* diagram.

2.1.3 Detailed Pattern view

The purpose of diagrams created under this view is to show the pattern in a more detailed view. At this level, we are able to view the details of the patterns and how it interfaces to all other design artifacts. The focus of this diagram is a pattern. The designer uses this diagram at a lower level to develop class diagrams.

Creation: A *Detailed Pattern* diagram can be viewed for each pattern in the *Pattern-Level* diagram.

Working Toolbar Artifacts: When editing a *Pattern-Level* diagram, the Working Toolbar shall contain the Notes and Note Connection.

Schematic Diagram: A *Detailed Pattern* diagram contains the following artifacts:

a) Detailed Patterns
The internal structure of the pattern can be viewed and the internals interconnection with the pattern interfaces can be revealed.

b) Connectors
Connectors are used to identify the interface to other model artifacts such as Interfaces, Patterns, or Subsystems. The specification window of the Connector shows other parts of the design the uses this Connector.

Figure 6, shows an example of a *Detailed Pattern* diagram for Pattern2.
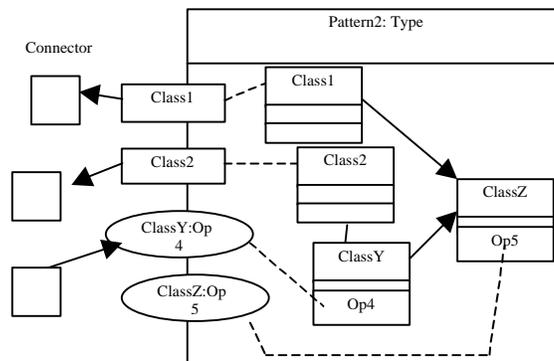
Figure 6. A detailed pattern diagram

2.2 Analysis and Design

2.2.1 Design Problems
Three problems need to be examined in the design of the GUI module.

Diagram Structure

The major responsibility of the GUI of the POAD Environment is to support creating and editing of diagrams. Diagram is the most important element in the GUI module. A diagram can be thought of as a drawing surface that symbols representing patterns, subsystems or dependency relations can be drawn upon. There are three levels of diagrams and each level can contain different elements. The following entities are essential to the three levels of diagrams.

Pattern

This entity encapsulate essential information about a design pattern —its name, its type, its interface and its internal structure. The interface of a pattern consists of methods and classes while the internal structure of the pattern is made up of classes. A pattern will have different visual representation when it appears in different level of diagram. In a pattern level diagram, it is represented by a rectangle with its instance name and type displayed inside the rectangle, with its interface and internal structure hidden; in a interface diagram, it is represented by a box. The interface of the pattern should be revealed at this level; in a detailed pattern diagram, it is also represented by a box, but both its interface and internal structure are revealed.

Subsystem

This entity captures essential information about a subsystem in a design. It is represented using a package symbol. The internal structure of a subsystem can be represented by a pattern level diagram. A subsystem can have dependency relation with another subsystem or pattern.

Dependency Relation

This entity represents the relationship between two patterns, two subsystems or a pattern and a subsystem. In the diagram, it is represented by dotted line directed from an entity to another entity. A dependency relation between two patterns can be expanded to reveal interface connection between the two patterns.

Interface Association

This entity represents the relation between two interface elements that each belongs to a different pattern. The interface element can be a class or a method.

Inter-class Relation

This entity represents the relation between classes in the internal structure of a pattern. The relation is the traditional class associations that can be: Aggregation, Dependency or Generalization.

Object Interactions

The GUI of the POAD Environment is a complex subsystem. To implement it using Object-Oriented approach, we need to distribute its behavior among a large number of objects. Such distribution can result in a structure with many connections between objects; in the worst case, every object ends up knowing about every other. We have to

consider how to control and coordinate object interaction to avoid a tightly coupled structure.

Integration Issues

The GUI module and the Design Engine are both components of the POAD Environment. They need to communicate with each other for the tool to function. Because the GUI is developed in advance of the Design Engine, we need to define the communication path between the two modules carefully and keep good documentation of it, so that task of integration could be easier in the next phase.

2.2.2 Top Level Design

Two patterns are adopted in the design of the GUI module. To control object interaction, collective behavior can be encapsulated in a separate mediator object. A mediator is responsible for controlling and coordinating the interactions of a group of objects. For details of this pattern, readers could refer to Appendix A. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections.

For easy integration of the GUI with the Design Engine, it is essential to restrict the communication path between the GUI module and the Design Engine. For this purpose, a Facade pattern be used as a communication link between the GUI and the Design Engine. The Facade pattern provides a single, simplified interface to the more general facilities of the Design Engine. The GUI module communicate with the Design Engine only through the Facade. Any user interaction that concerns the Design Engine will cause a message to be sent to the Facade. The Facade responds to the messages by executing methods. Initially, the methods in the Facade may only have an empty body. These methods can be implemented later in the implementation of the Design Engine. Again, for details of the Facade pattern, readers could refer to Appendix A.

Based upon the above rational, a top level design for the interface is presented below using a pattern level diagram, which is generated by the POAD environment. In the diagram, UIDirector is shown as an instance of Mediator pattern and facade is shown as an instance of Facade pattern. Standard Control, Browser, Graphics Editor, and Documentation Viewer are shown as subsystems.
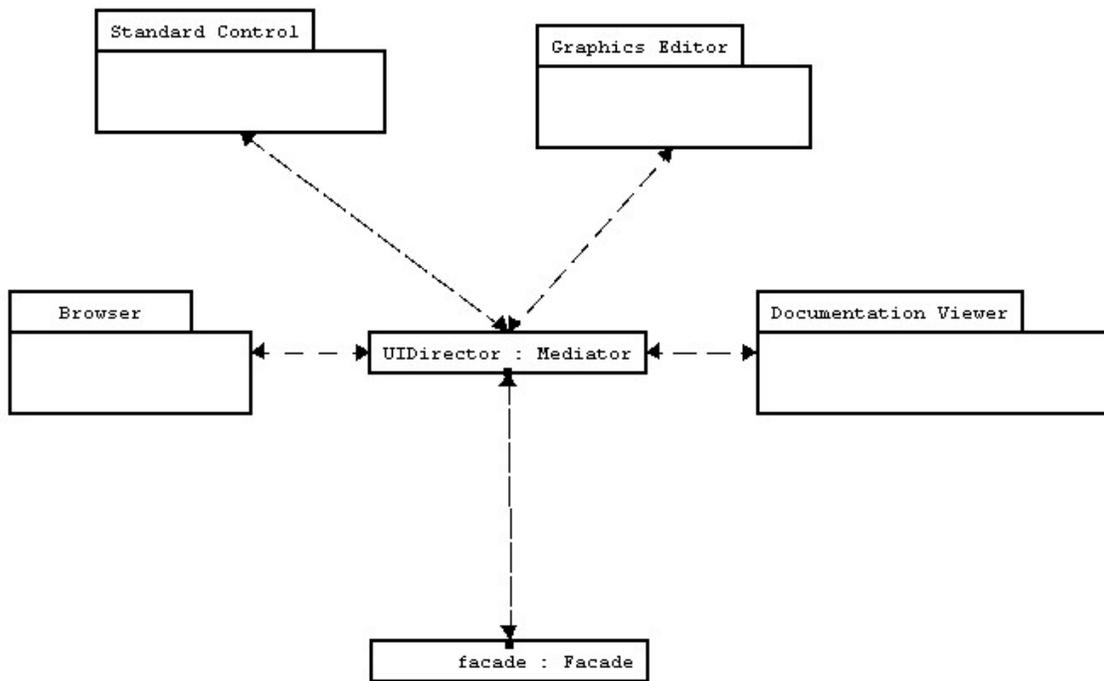
Figure 7. Top level design of the GUI

Standard Control Subsystem

The Standard Control Subsystem consists of menus and toolbar that are somewhat standard to application with GUI. It is responsible for basic I/O functionality such as opening file and saving file and basic editing functionality such as cut, copy and paste.

Browser Subsystem

The Browser Subsystem consists of a Diagram Browser that enables a user to organize multiple diagrams easily. The Diagram Browser will keep a list of opened diagrams organized into three levels of views. A user can click on any diagram and bring it to foreground.

Documentation Viewer Subsystem

The Documentation Viewer Subsystem consists of a window that displays the documentation for a selected item in a diagram.

Graphics Editor Subsystem

The Graphics Editor Subsystem consists of a set of diagrams, a working toolbar and a desktop like window that is used to organize diagrams. It is the most complex and most important subsystem in the GUI module. Most of the GUI's functionality is realized in this subsystem.

UIDirector

The UIDirector is modeled after the Mediator pattern. It knows about all the other subsystems and acts like a liaison between them. All communications between other subsystems will pass through UIDirector.

Facade

The Facade provides a unified interface for the design engine. It is the connection link between the GUI and the design engine. All messages from GUI components that concerns the Design Engine will eventually be sent here. The Facade responds to messages by executing methods. Initially many methods may only have an empty body, which can be implemented later in the development phase of the Design Engine.

2.2.3 Detailed Design

The Browser, Documentation Viewer, Standard Control, Facade and UIDirector are simple subsystems and may be implemented using a few objects. However, Graphics Editor is much more complex. What makes this subsystem complex is that it must support editing and management of multiple diagrams, each diagram can have very different behavior and diagrams have to communicate with each other and with the working toolbar. A design diagram for the Graphics Editor subsystem is presented below.
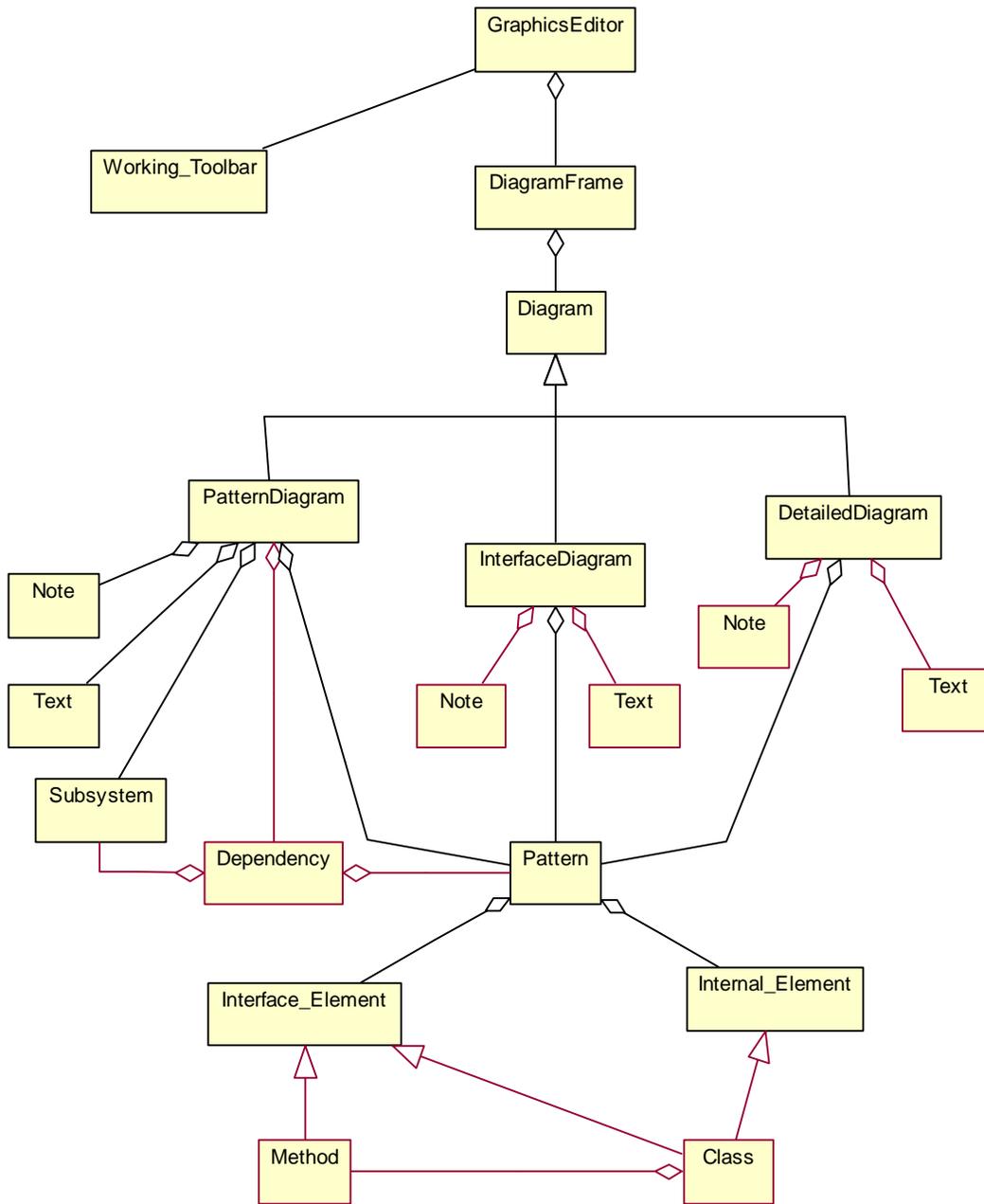
Figure 8. Detailed design of the Graphics Editor Subsystem

As can be seen from the diagram, the Graphics Editor subsystem consists of many components, many of which need to communicate with each other and with the UIDirector. Based on the same rational for using UIDirector as the mediator for the entire GUI module, a mediator is needed for the Graphics Editor subsystem. The object GraphicsEditor plays this role. It is named with the name of the subsystem because the other subsystems communicate with the Graphics Editor subsystem through this object. The GraphicsEditor controls and coordinates the interaction among objects in the

Graphics Editor subsystem. For example, when a diagram is created or activated (brought to the foreground), the diagram will send a message to the GraphicsEditor asking it to change the content of the working toolbar accordingly. When an object need to communicate with the Facade, it must first send a request to the GraphicsEditor, which in turn send a request to the UIDirector. The UIDirector finally delegates the request to the Facade.

A complete detailed design is obtained by putting the design of the Graphics Editor subsystem with designs of other subsystems together.
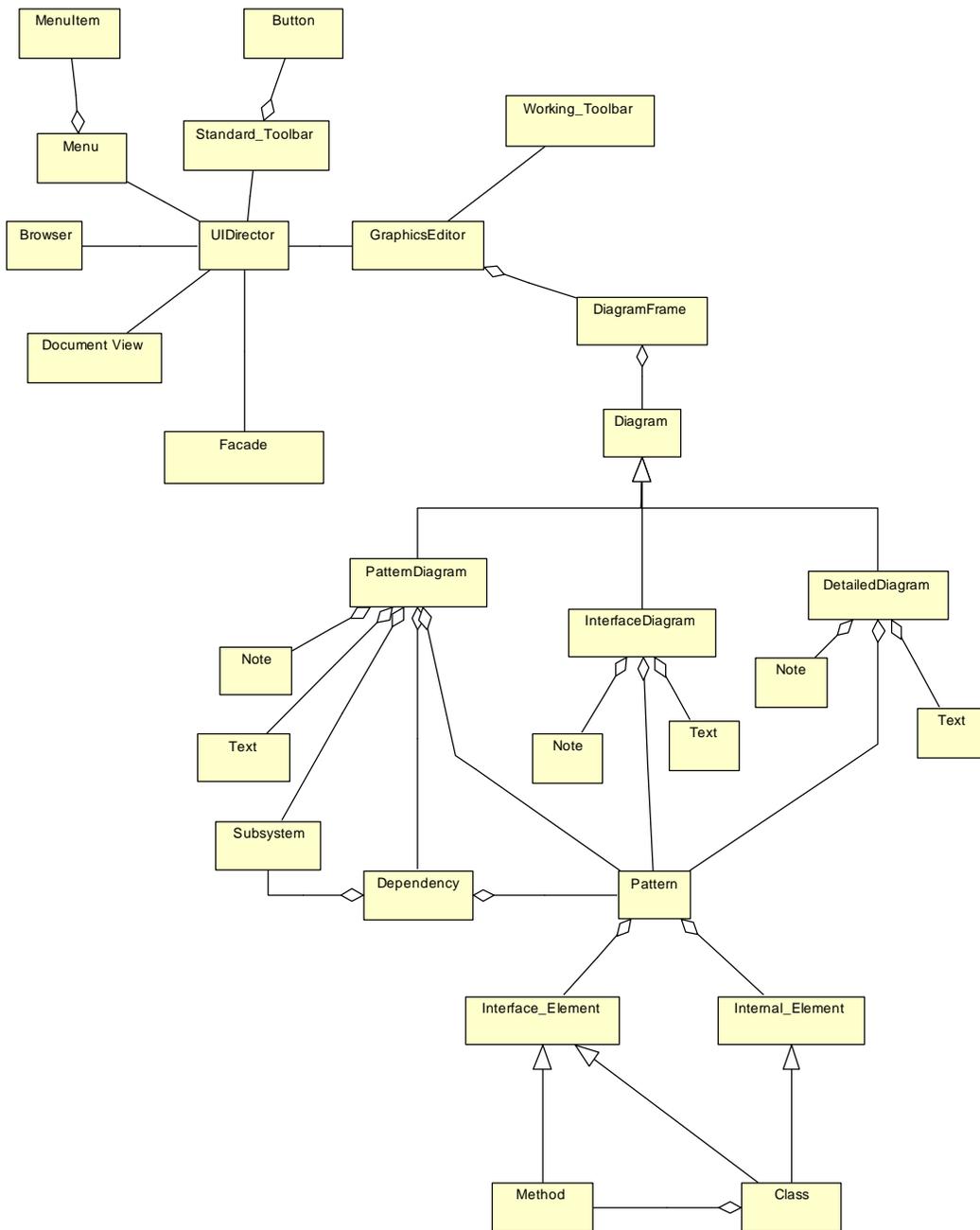
Figure 9. Detailed design of the GUI module

The following is a brief description of each object involved in the design.

Browser

The Browser organizes the list of diagrams in three level of views. Each view is represented by a folder and the diagrams in that view is represented as the nodes of the folder. The Browser only knows about the UIDirector. If a user double clicks a node, the

Browser will send a message to the UIDirector asking it to show the diagram corresponding to the node in the foreground. The communication between the Browser and UIDirector is bi-directional. If a new diagram is created, the UIDirector will ask the Browser to add a node to the appropriate folder.

Documentation Viewer

The Documentation Viewer displays text on behalf of the UIDirector. The text is usually documentation associated with an item in a diagram. The communication between the UIDirector and Documentation Viewer is uni-directional. The Documentation Viewer won't send any message to the UIDirector.

Button

The Button is contained in a standard toolbar to realize a certain functionality. It only knows about the UIDirector. If it is clicked, it sends a message to the UIDirector. The communication is bi-directional. The Button could be disabled by the UIDirector.

MenuItem

The MenuItem is an item in a pull down menu that is used to realize certain funcationality. It only knows about the UIDirector. If it is clicked, it sends a message to the UIDirector. The communication is bi-directional. The MenuItem could be disabled by the UIDirector if it is context sensitive.

UIDirector

The UIDirector is the top level mediator for the interface. It knows about all the other components and it coordinates behavior and apperance of the entire interface.

Facade

The Facade is where the GUI is connected to the application. Functionality of the design engine is abstracted and provided in the Facade with a convenient interface.

GraphicsEditor

The GraphicsEditor is the second level mediator. It coordinates the behavior and appearance of the Graphics Editor module. It knows about every gadgets in the Graphics Editor subsystem and the UIDirector. The communication between the UIDirector and GraphicsEditor is bi-directional.

Working Toolbar

One toolbar is defined for each levels of diagram. Each toolbar contains a set of buttons that can be used to change the behavior of the diagram. All the toolbars only know about the GraphicsEditor. The communication is bi-directional.

Diagram

The Diagram is a drawing surface that symbols for patterns, subsystems and notes, etc. can be drawn on. It has a drawing tool that is represented by the mouse pointer. The cursor for the mouse pointer takes on different shape for each setting of the drawing tool. PatternDiagram, InterfaceDiagram and DetailedDiagram are derived from Diagram.

DiagramFrame

The DiagramFrame is the container for a Diagram. Each DiagramFrame opens a separate window so that all diagrams can be managed efficiently. The DiagramFrame also supports scrolling of the Diagram.

PatternDiagram

The PatternDiagram implements a drawing surface for a pattern level diagram. It is derived from Diagram.

InterfaceDiagram

The InterfaceDiagram implements a drawing surface for an interface level diagram. It is derived from Diagram.

DetailedDiagram

The DetailedDiagram implements a drawing surface for a detailed pattern diagram. It is derived from Diagram.

Pattern

The Pattern is the internal representation of a design pattern in the program. A Pattern object has a instance name, pattern type, documentation, an interface made up of classes and operations, and an internal structure. On a diagram, a pattern is represented by a symbol. The symbol for a pattern is different on different types of diagram.

Subsystem

The Subsystem is the internal representation of a subsystem in the program. A Subsystem object has a name, documentation and a symbol.

Interface_Element

The Interface_Element is the internal representation of an element that makes up the interface of a pattern. An Interface_Elemment can be a Class or Method.

Internal_Element

The Internal_Element is the internal representation of an element that makes up the internal structure of a pattern.

Class

The Class object is a subclass of both the Interface_Element and Internal_Element. It is an element of the internal structure of a pattern and it can optionally show up in the pattern's interface.

Method

The Method object is a subclass of Interface_Element. It is owned by a Class which is an internal element of the pattern.

2.3 Implementation

The GUI of the tool is implemented in Java using Swing and Java 2D graphics. Unlike Java AWT, which depends upon native peers, Swing components are written entirely in Java and don't depend on the native environment. Applications written using Swing components can be expected to be portable among multiple platforms. Java 2D graphics allows a Java programmer to construct complex geometric shapes from primitives ones. Composite shapes can be represented using a single object, making the task of representing different items on the diagram much easier and the design cleaner.

An attractive new feature of Swing is the introduction of actions [14]. An action allows a programmer to bundle a commonly used procedure and its bound properties (including an image to represent it) into a single class. Actions are often used to implement menu items or buttons. In this case, an action will store properties about the menu item or button as well as event handling routine. It then can be added to many Swing containers. When the action is placed in a menu, a menu item will be created using the label or image bundled with the action, then the action is registered as the event listener for the menu item. The same holds if the action is placed in a toolbar. Both the menu item and the button will reference the same action object. If the function associated with the action should be disabled for some reason, the property can be set in the action. The menu item and button are automatically notified that they can no longer be selected or pressed and can relay that information to the user. The action construct is heavily used in the implementation of Standard Control subsystem.

Most of the objects presented in the detailed design are implemented with a equivalent Java class. Readers could refer to the source code included in Appendix D or go to http://www.csee.wvu.edu/~xue/source.html for more details.

The following picture is a snapshot of the GUI running under Windows 98.
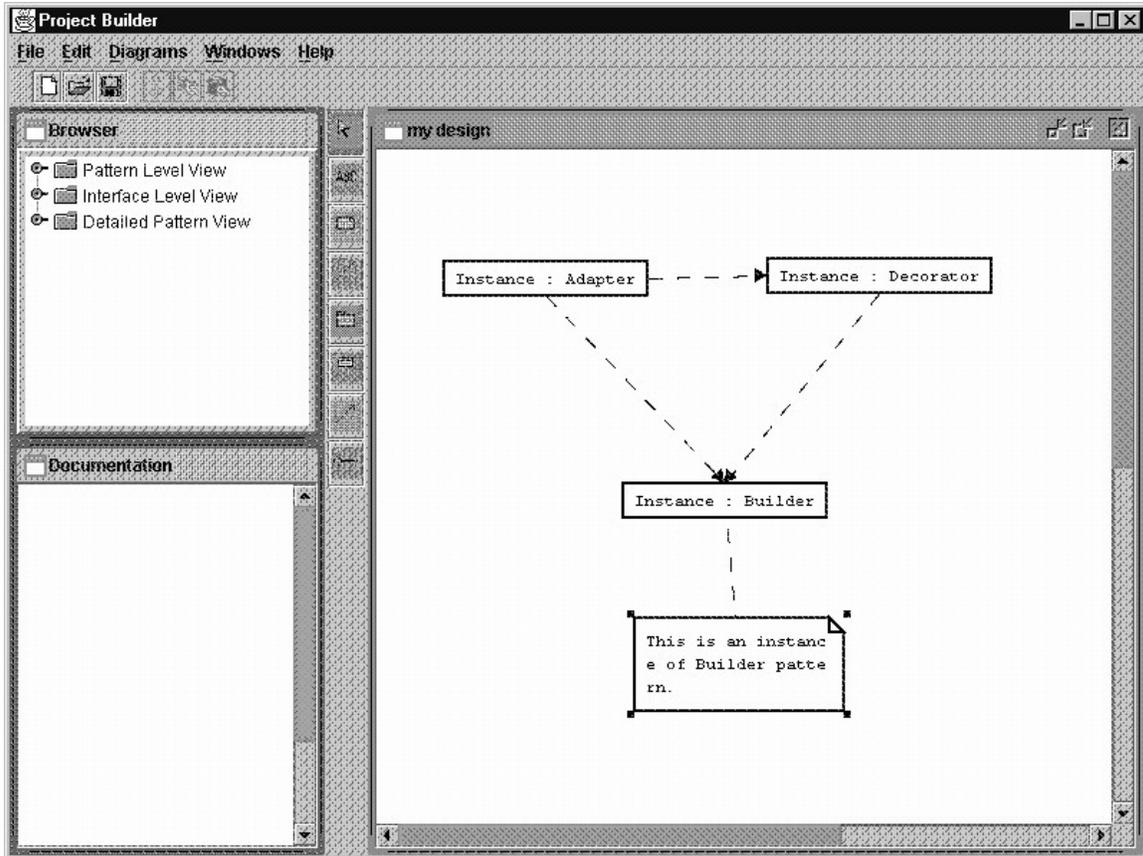
Figure 10. A snapshot of the GUI

3. DEVELOPMENT OF THE DATABASE

The database holds the repository of constructional design patterns available as reusable design components. For each pattern, we store conventional elements such as name, intent, alias, motivation, etc. An internal representation of the pattern interface and pattern internal structure will also be saved as the pattern model so that a pattern can be readily used in a design. For easy browsing and searching of the database, we also associate keywords with each pattern. Related patterns are organized into categories or classifications. Entity-Relation model and actual table definitions for the database are presented in the following two subsections.

3.1 Entity-Relation modeling

The two entities involved in the database are patterns and pattern categories. The following E-R diagram describe their relation.
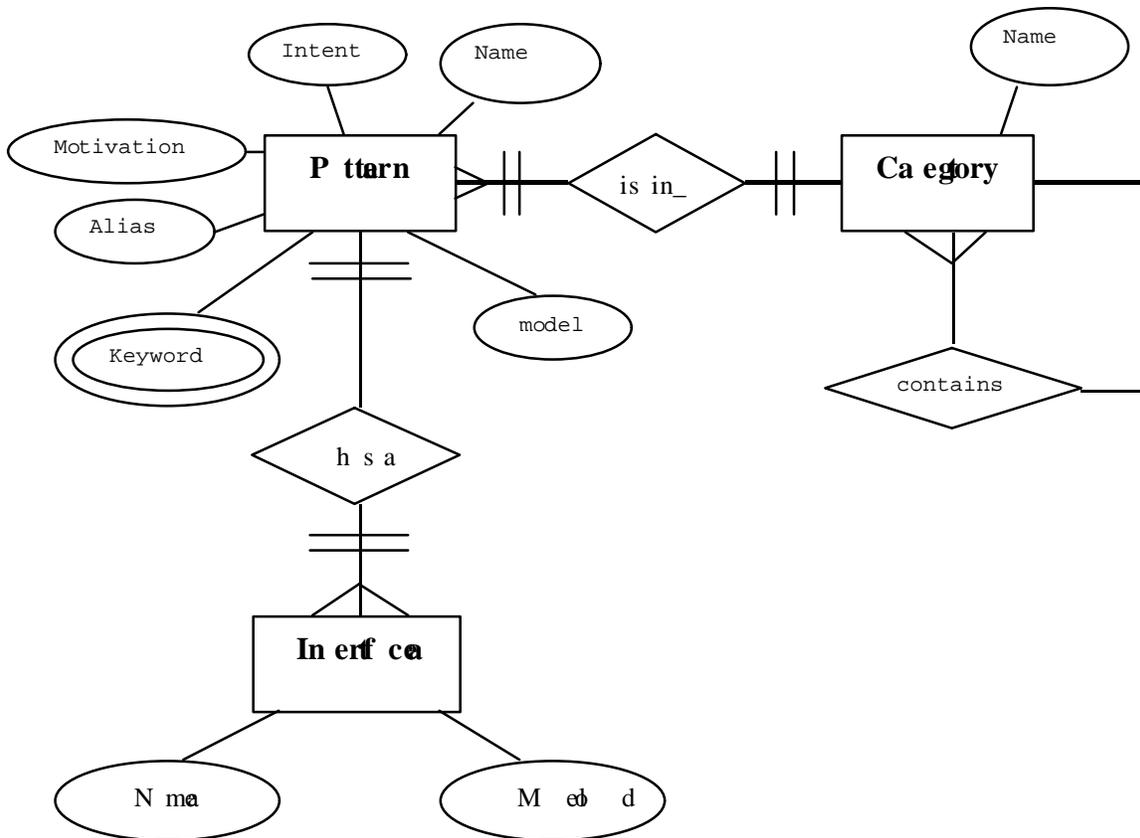
Figure 11. E-R diagram for the database

In the diagram, some of the attributes of Pattern entity were omitted to save space. The complete attributes set for the Pattern entity are: name, category, intent, aliases, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, related patterns , model, interface and keyword.

3.2 Table definitions

The E-R model presented in the 3.1 is translated into relational model. Four tables are defined-Patterns, Categories, Keywords and Interfaces. The definition for each table is presented below.

Patterns table definition

Attribute	Data Type	Constraint
name	String	Primary Key
category	String	Foreign Key
intent	String	-
aliases	String	-
motivation	String	-
applicability	String	-
structure	Binary File	-
participants	String	-
collaborations	String	-

consequences	String	-
implementation	String	-
sample_code	String	-
known_uses	String	-
related_patterns	String	-
model	String	-

Categories table definition

Attribute	Data Type	Constraint
name	String	Primary Key
parent_category	String	References name

Keywords table definition

Attribute	Data Type	Constraint
keyword	String	
pattern_name	String	Foreign key

* keyword + pattern_name forms primary key

Interfaces table definition

Attribute	Data Type	Constraint
name	String	
pattern_name	String	Foreign key
model	String	

* name + pattern_name forms primary key

4. DEVELOPMENT OF THE PATTERN BROWSER

4.1 Requirements

The Pattern Browser is used to browse the database of constructional design patterns. The pattern browser should
- be implemented as a Java Applet and be able to run inside a web browser.
- be able to connect to the database through the Server Application using JDBC.

The interface of the applet should
- composed of a browsing window that has a tree structure of patterns under pre-defined categories and another window that displays details of patterns.
- let a user view the details of a pattern by double clicking on a pattern name in the browser window.
- displays a frame for each pattern with folder structure.
- display related patterns as hyperlinks which a user can use to jump to another pattern.
- allow a user to search the pattern database by specifying one or more keywords.

4.2 Analysis and Design

Design Problems

Three problem need to be examined when designing the Pattern Browser.

How to minimize dependency on the database.

Because the Pattern Browser will connect directly to the database, we are actually using a two-tiered architecture. It is inevitable that the Browser will depend upon the database. The problem is to minimize this dependency and isolate the knowledge of the database to as few objects as possible so that when the database changes, only small part of the Browser need to be changed.

How to convert relational data into objects

To build a front end to a relational database using object-oriented approach, we need a mapping of the data from the relation world to object-oriented world. Once again, this mapping should affect minimal number of objects and be invisible from the rest of the Pattern Browser. In case tables schemas changes in the database, we only need to modify small part of the Pattern Browser.

How to achieve efficiency

Retrieving data from database is a time consuming process. If every data the Pattern Browser requires needs to be brought in from the database, the speed will be intolerable. An efficient design should keep a small capacity cache at the front end. Anything already in the cache could be presented to the user instantaneously. Only data not in the cache need to be brought in from the database.

Top Level Design

The Pattern Browser is divided into three major components as illustrated in the following pattern level design diagram.
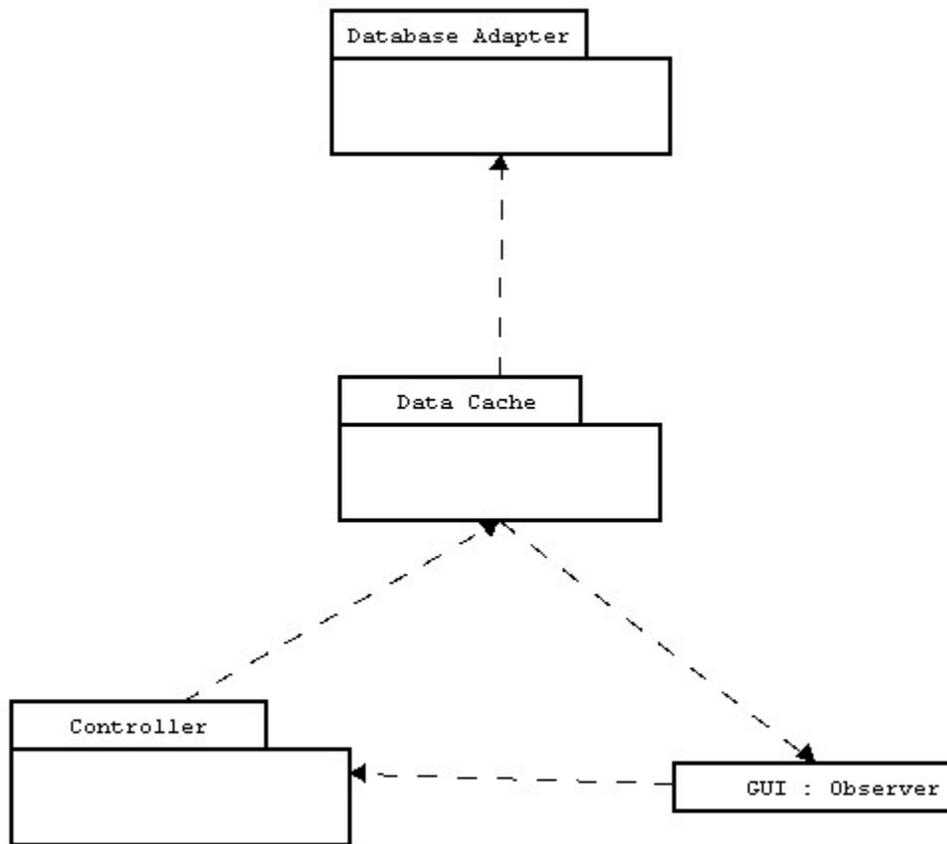
Figure 12. High level design of the Pattern Browser

Database Adapter

This component is responsible for making connection to the database and construct object from relatively primitive data from the database. All knowledge of the database is isolated in this module.

Data Cache

This component acts as a temporary data store for information retrieved from the database. It tries to satisfies user's request using its own data. Only when the data requested is not in the store will it go to the Database Adapter module for new data.

GUI

This component is responsible for constructing the graphical user interface for the Pattern Browser. It is an instance of Observer pattern.

Controller

This component is responsible for processing user interaction originated in the GUI component and making decisions on what to do.

The Data Cache, GUI, Controller components and their interactions are designed according to the Model-View-Controller (MVC) structure. Here the Data Cache act as the Model and the GUI acts as the View.

Detailed Design

A more detailed design is obtained by expanding the four components in the top level design. The following diagram shows the objects involved and their relations.
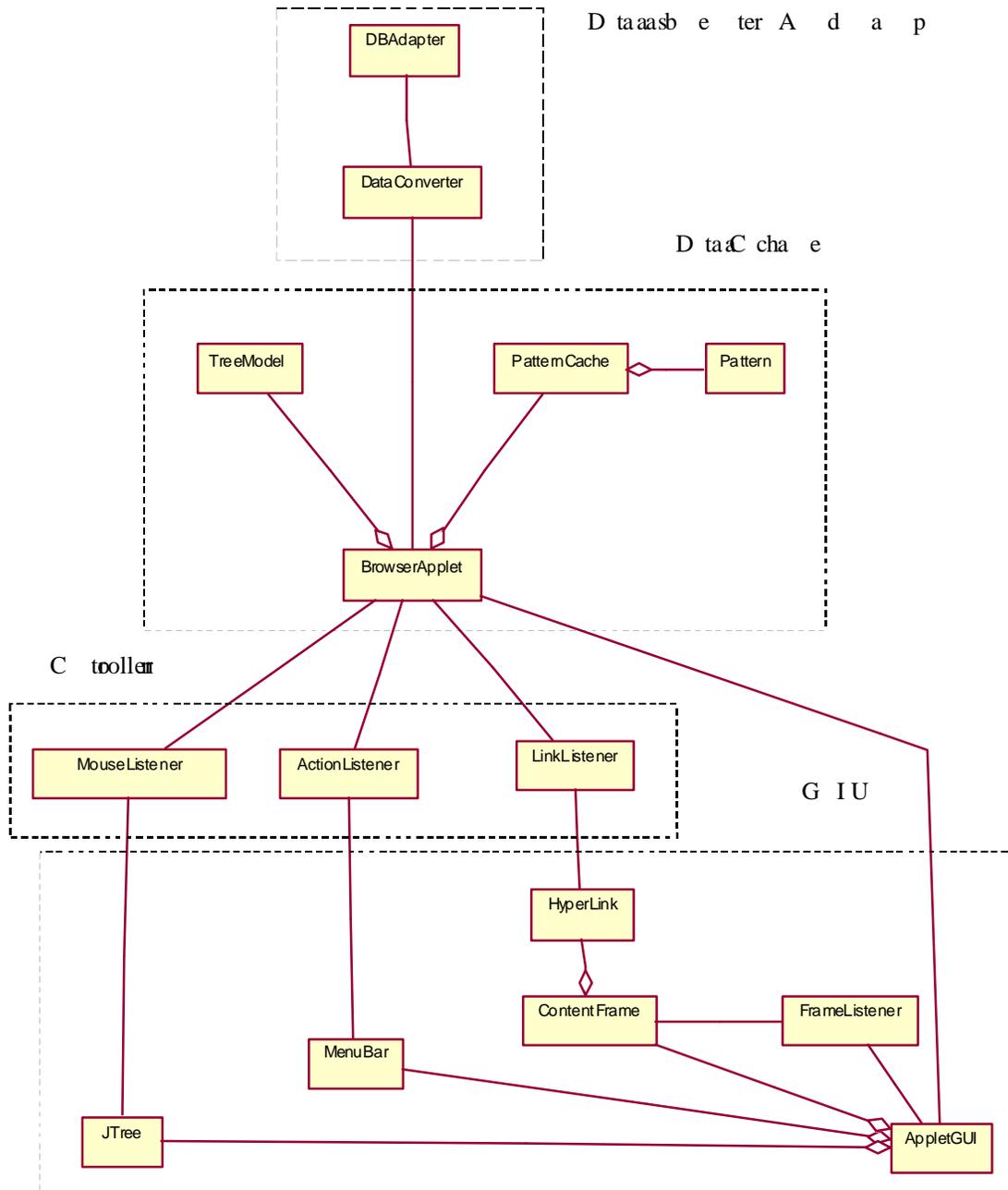
Figure 13. Detailed design for the Pattern Browser

DBAdapter

This object is responsible for making connection to the database and send SQL statements to the database through JDBC driver. It is the only object that has knowledge of the database.

DataConverter

This object is responsible for converting primitive data retrieved from the database into high level objects. More specifically, it constructs a tree like structure from the hierarchical structure made up by categories and patterns and a pattern object from attributes of the pattern in the database.

TreeModel

This object is named using Java Swing convention. It is a tree like construct that represents the hierarchical structure made up by different levels of categories and patterns. It serves as the index for the database.

Pattern

This Pattern object stores essential information about a design pattern. Its name, intent, motivation, etc.

PatternCache

This object keeps temporary storage of Patterns already retrieved from the database.

BrowserApplet

This object is a subclass of Applet and is the entry point for the whole Pattern Browser. It keeps references of the TreeModel and PatternCache

AppletGUI

This object is responsible for constructing graphical user interface for the applet. It displays the TreeModel using a JTree object and a Pattern using JTabbedPane. It also has a menubar that contains menu items for searching the database and managing windows.

MouseListener

This object listens to mouse event originated from the JTree object in the AppletGUI. It the mouse is double clicked on a leaf, this object will send a request to the BrowserApplet to get the Pattern represented by the leaf.

ActionLsitener

This object listens to action event originated from menu items contained in the AppletGUI.

LinkListener

This object listens to event originated from hyperlinks embedded in the documentation for a certain pattern. Usually the hyperlink refers to another pattern. This object will send a request to the BrowserApplet to get the pattern referred to by the hyperlink.

ContentFrame

This object display details about a Pattern. It contains a tabbed pane. Each tab displays one attribute of the Pattern.

FrameListener

This object listens to event originated from the ContentFrame. It allows a user to switch from a ContentFrame to another ContentFrame by mouse clicking. Because this type of event only changes the appearance of the user interface, it is kept as part of the GUI rather than Controller

MenuBar

This object implements the menu bar for the Pattern Browser. It contains a 'Search' menu and a 'Windows' menu. 'Search' menu is for searching the database while 'Windows' menu is for managing internal windows of the applet.

4.3 Implementation

The Pattern Browser is implemented as a Java applet using Java Swing Components and JDBC classes. The advantage of using Swing Components discussed in section 3 is also valid here. JDBC is a SQL-level API that allows a programmer to embed SQL statements as arguments to methods in JDBC interfaces [15]. To allow a programmer to do this in a database-independent fashion, JDBC requires database vendors to furnish a run-time implementation of its interfaces. There implementations route SQL calls to the database in the proprietary fashion it recognizes. The programmer, however, never need to worry about how the SQL statements are routed. The facade provided by JDBC gives a programmer complete freedom from any database specific issues-the same code can run no matter what database is present.

The objects presented in the detailed design are implemented using Java classesReaders could refer to the source code included in Appendix D or go to http://www.csee.wvu.edu/~xue/source.html for more details.

The following picture shows a snapshot of the Pattern Browser running with a Macintosh Look&Feel.
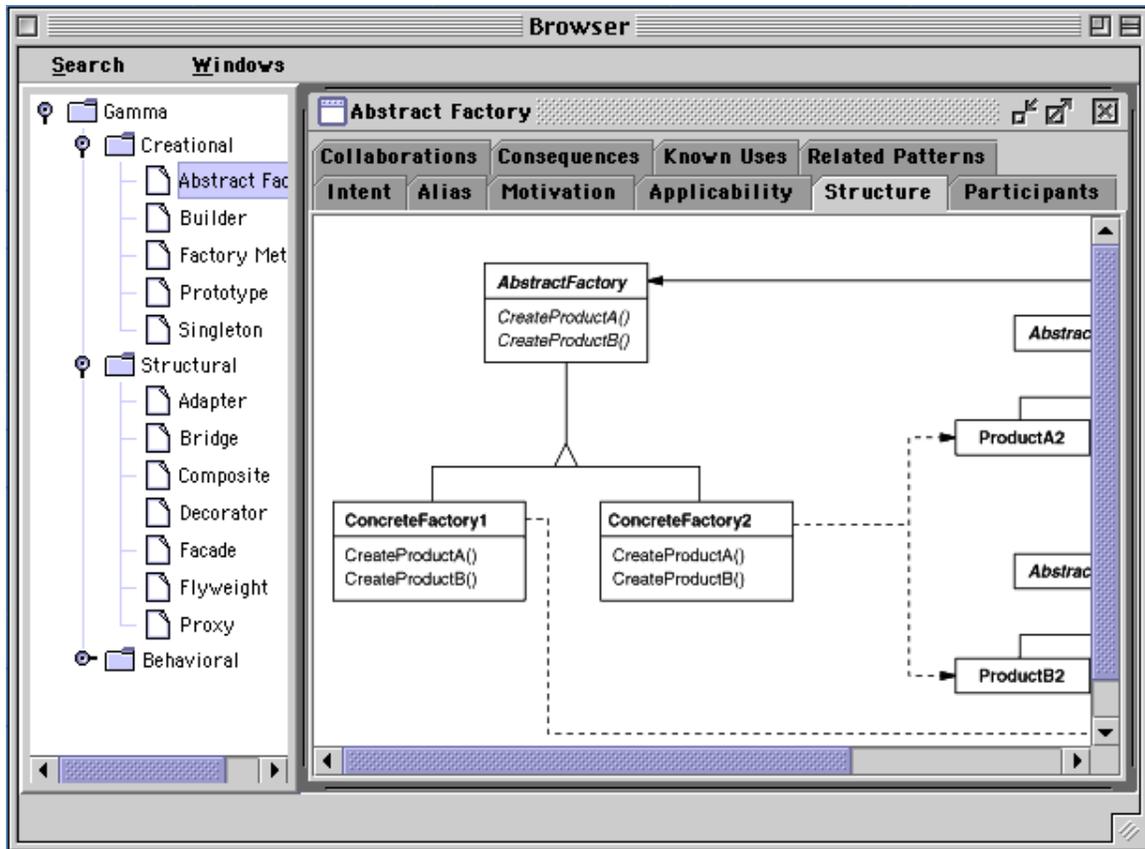
Figure 14. A snapshot of the Pattern Browser

5. CONCLUSION

The tool is useful in hierarchical design in that it encourages reuse of previously documented design patterns in designing reusable architectures that are easily traced to object oriented designs at lower design levels. The tool facilitates the development of architectures that are hierarchical and traceable to lower level design while still preserving architecture view of the application. At this stage, the database, the Pattern Browser and user interface of the POAD Environment have been successfully developed. Patterns can be added to the database and remote accessed by the Pattern Browser. The GUI of the tool supports Pattern Level Diagram, Pattern Interface Diagram and Detailed Pattern Diagram. All notations in the diagram are compatible with UML standards. The next development phase for the tool is to add the model checker capability to determine interface mismatches and constraints checks; i.e. *Model Checker* component. and apply the tool to develop OO design frameworks for distributed object applications such as medical informatics systems.

6. REFERENCES

[1]   Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Object-Oriented Software", Addison-Wesley, 1995

[2]   Quatrani, T., "Visual Modeling with Rational Rose and UML", Addison-Wesley, 1998

[3]   Yacoub, S. M., H. Ammar, "Object Oriented Design Patterns as Design Components"

[4]   Yacoub, S. and H. Ammar, "Tool Support for Developing Pattern-Oriented Architectures", Proceedings of the 1st Symposium on Reusable Architectures and Components for Developing Distributed Information Systems, RACDIS'99, Orlando, Florida, August 2-3, 1999, pp665-670

[5]   Meijler , T. D., S. Demeyer, and R. Engel, "Making Design Patterns Explicit in FACE, A Framework Adaptive Composition Environment", in Software Engineering Notes, ESEC/FSE, Vol. 22, No 6, Nov 1997, pp94-110.

[6]   Gert Florijin, Marco Meijers, Pieter van Winsen, "Tool support for Object-Oriented Patterns", Proceedings of the European Conference of Object Oriented Programming, ECOOP'97, p472.

[7]   Schuetze, M., J. P. Riegel, and G. Zimmermann, "A Pattern-Based Application Generator for Building Simulation", in Software Engineering Notes, ESEC/FSE, Vol. 22, No. 6 Nov 1997, pp468-482

[8] Budinsky, F., M. Finnie, J. Vlissides, and P. Yu, "Automatic Code Generation from Design Patterns," IBM Systems Journal, Vol 35, No. 2, 1996.

[9]   Sefika, M., A. Sane, R. Campbell, "Monitoring Compliance of a Software System with its high-Level Design Models", Proc. Of ICSE'96, 1996.

[10]   Pagel, B., and M. Winter, "Towards Pattern-Based Tools", EuroPLoP Preliminary Conference Proceedings, July 1996.

[11]   Blueprint   Inc.,   Framework   Studio,   http://www   blueprint-technologies.com/product/framework studio/index.html

[12]   The   Unified   Modeling   Language   Resource   Center http://www.rational.com/uml/index.html

[13]   Yacoub, S. and H. Ammar, "Towards Pattern Oriented Frameworks", to appear in Journal of Object Oriented Programming JOOP, 1999

[14]   Eckstein, R., M. Loy and D. Wood, "Java Swing", O'reilly, 1998

[15]   Reese, G., "Database Programming with JDBC and Java", O'Reilly & Associates, Inc., 1997

APPENDIX

A. Design Patterns used in the report

The design patterns Facade, Mediator, and Observer were deployed in the design of the GUI of the POAD environment and the pattern browser. The details of these patterns were taken from [1] and listed here in the standard template format to serve as references for the readers.

1. Facade

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.
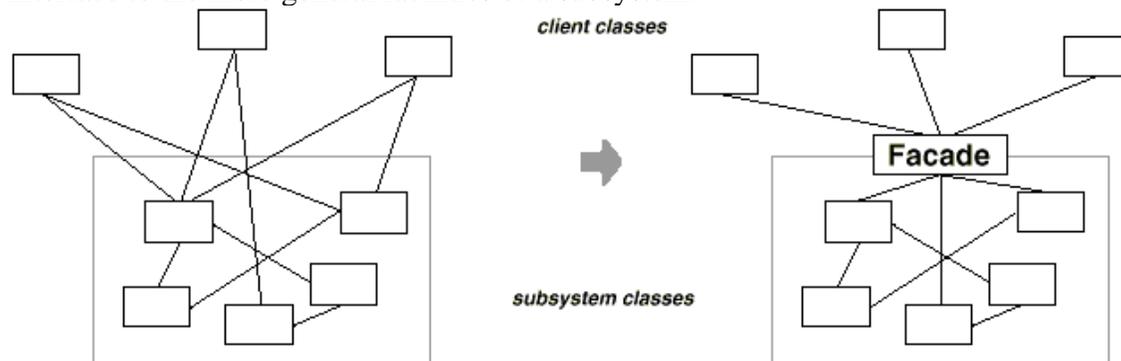
Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler. Some specialized applications might need to access these classes directly. But most clients of a compiler generally don't care about details like parsing and code generation; they merely want to compile some code. For them, the powerful but low-level interfaces in the compiler subsystem only complicate their task.

To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class. This class defines a unified interface to the compiler's functionality. The Compiler class acts as a facade: It offers clients a single, simple interface to the compiler subsystem. It glues together the classes that implement compiler functionality without hiding them completely. The compiler facade makes life easier for most programmers without hiding the lower-level functionality from the few that need it.
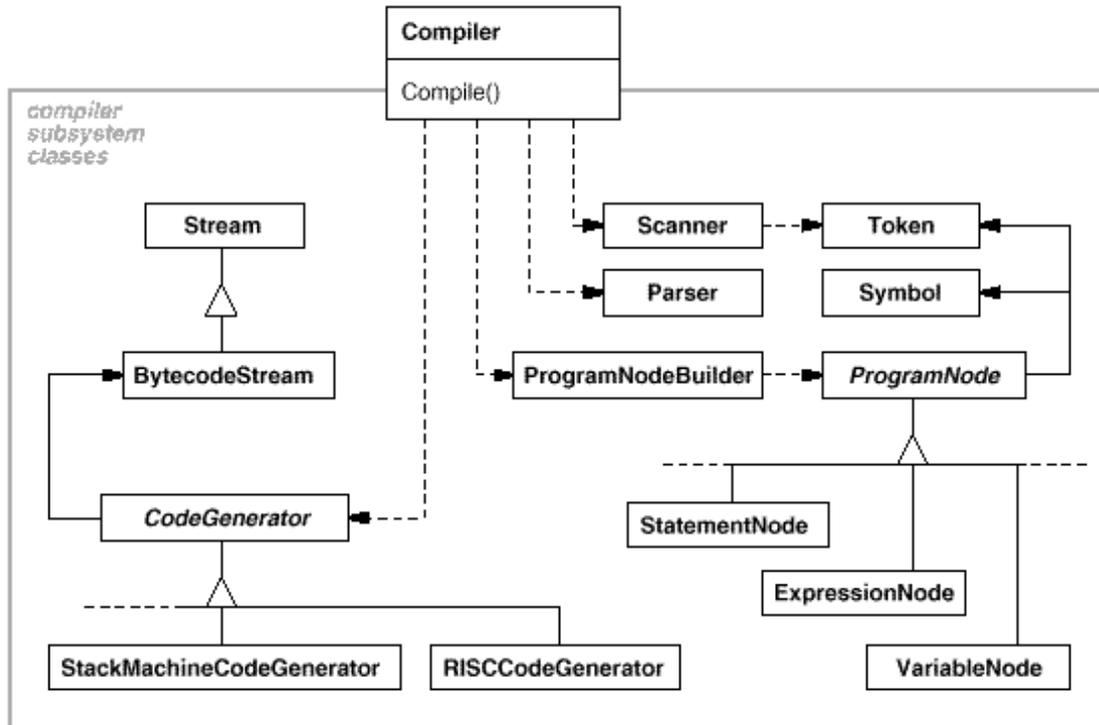
Applicability

Use the Facade pattern when

- ♦ you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier

- ♦ there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.

- ♦ you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Structure

Collaborations

- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- Clients that use the facade don't have to access its subsystem objects directly.

Participants

- Facade (Compiler)
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- subsystem classes (Scanner, Parser, ProgramNode, etc.)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

Consequences

The Facade pattern offers the following benefits:

1.It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.

2.It promotes weak coupling between the subsystem and its clients. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies. This can be an important consequence when the client and the subsystem are implemented independently.

Reducing compilation dependencies is vital in large software systems. You want to save time by minimizing recompilation when subsystem classes change. Reducing compilation dependencies with facades can limit the recompilation needed for a small change in an important subsystem. A facade can also simplify porting systems to other platforms, because it's less likely that building one subsystem requires building all others.

3.It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Implementation

Consider the following issues when implementing a facade:

1.Reducing client-subsystem coupling. The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.

An alternative to subclassing is to configure a Facade object with different subsystem objects. To customize the facade, simply replace one or more of its subsystem objects.

2.Public versus private subsystem classes. A subsystem is analogous to a class in that both have interfaces, and both encapsulate something—a class encapsulates state and operations, while a subsystem encapsulates classes. And just as it's useful to think of the public and private interface of a class, we can think of the public and private interface of a subsystem.

The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. The Facade class is part of the public interface, of course, but it's not the only part. Other subsystem classes are usually public as well. For example, the classes Parser and Scanner in the compiler subsystem are part of the public interface.

Making subsystem classes private would be useful, but few object-oriented languages support it. Both C++ and Smalltalk traditionally have had a global name space for classes. Recently, however, the C++ standardization committee added name spaces to the language, which will let you expose just the public subsystem classes.

Known Uses

The compiler example in the Sample Code section was inspired by the ObjectWorks\Smalltalk compiler system.

In the ET++ application framework, an application can have built-in browsing tools for inspecting its objects at run-time. These browsing tools are implemented in a separate subsystem that includes a Facade class called "ProgrammingEnvironment." This facade defines operations such as InspectObject and InspectClass for accessing the browsers.

An ET++ application can also forgo built-in browsing support. In that case, ProgrammingEnvironment implements these requests as null operations; that is, they do nothing. Only the ETProgrammingEnvironment subclass implements these requests with operations that display the corresponding browsers. The application has no knowledge of whether a browsing environment is available or not; there's abstract coupling between the application and the browsing subsystem.

The Choices operating system uses facades to compose many frameworks into one. The key abstractions in Choices are processes, storage, and address spaces. For each of these abstractions there is a corresponding subsystem, implemented as a framework, that supports porting Choices to a variety of different hardware platforms. Two of these subsystems have a "representative" (i.e., facade). These representatives are FileSystemInterface (storage) and Domain (address spaces).

For example, the virtual memory framework has Domain as its facade. A Domain represents an address space. It provides a mapping between virtual addresses and offsets into memory objects, files, or backing store. The main operations on Domain support adding a memory object at a particular address, removing a memory object, and handling a page fault.

As the preceding diagram shows, the virtual memory subsystem uses the following components internally:

- MemoryObject represents a data store.
- MemoryObjectCache caches the data of MemoryObjects in physical memory. MemoryObjectCache is actually a Strategy that localizes the caching policy.
- AddressTranslation encapsulates the address translation hardware.

The RepairFault operation is called whenever a page fault interrupt occurs. The Domain finds the memory object at the address causing the fault and delegates the RepairFault operation to the cache associated with that memory object. Domains can be customized by changing their components.

Related Patterns

Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.

Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.

Usually only one Facade object is required. Thus Facade objects are often Singletons.

2. Mediator

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objectsfrom referring to each other explicitly, and it lets you vary their interaction independently.

Motivation

Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about every other.

Though partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again. Lots of interconnections make it less likely that an object can work without the support of others—the system acts as though it were monolithic. Moreover, it can be difficult to change the system's behavior in any significant way, since behavior is distributed among many objects. As a result, you may be forced to define many subclasses to customize the system's behavior.

As an example, consider the implementation of dialog boxes in a graphical user interface. A dialog box uses a window to present a collection of widgets such as buttons, menus, and entry fields, as shown here:


Often there are dependencies between the widgets in the dialog. For example, a button gets disabled when a certain entry field is empty. Selecting an entry in a list of choices called a list box might change the contents of an entry field. Conversely, typing text into the entry field might automatically select one or more corresponding entries in the list box. Once text appears in the entry field, other buttons may become enabled that let the user do something with the text, such as changing or deleting the thing to which it refers.

Different dialog boxes will have different dependencies between widgets. So even though dialogs display the same kinds of widgets, they can't simply reuse stock widget classes; they have to be customized to reflect dialog-specific dependencies. Customizing them individually by subclassing will be tedious, since many classes are involved.

You can avoid these problems by encapsulating collective behavior in a separate mediator object. A mediator is responsible for controlling and coordinating the interactions of a group of objects. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections.

For example, FontDialogDirector can be the mediator between the widgets in a dialog box. A FontDialogDirector object knows the widgets in a dialog and coordinates their interaction. It acts as a hub of communication for widgets:

The following interaction diagram illustrates how the objects cooperate to handle a change in a list box's selection:
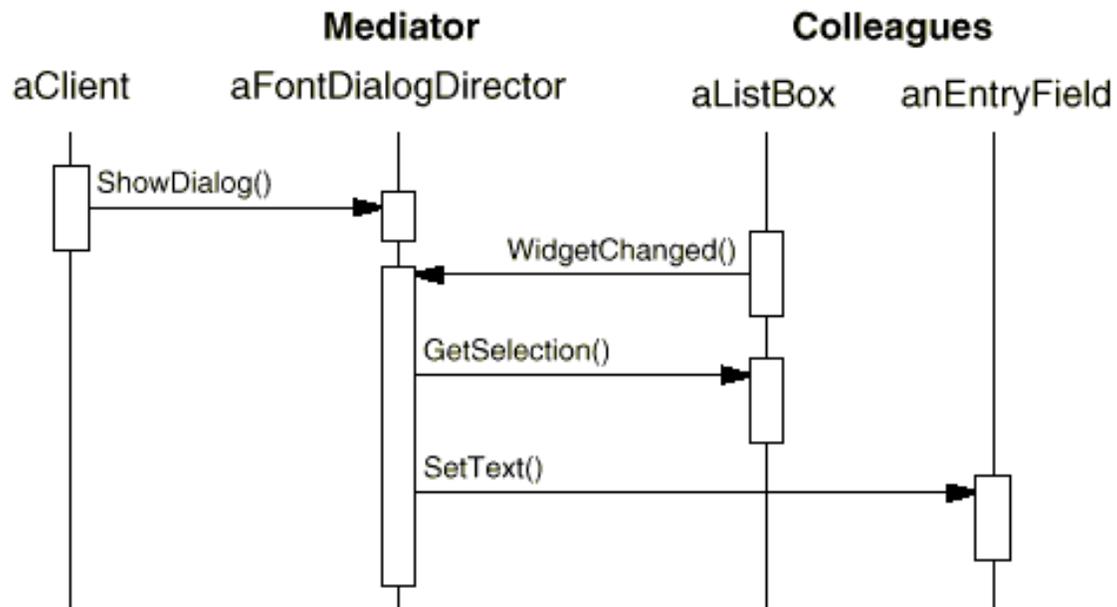

Here's the succession of events by which a list box's selection passes to an entry field:
    1.The list box tells its director that it's changed.
    2.The director gets the selection from the list box.
    3.The director passes the selection to the entry field.
    4.Now that the entry field contains some text, the director enables button(s) for initiating an action (e.g., "demibold," "oblique").

Note how the director mediates between the list box and the entry field. Widgets communicate with each other only indirectly, through the director. They don't have to know about each other; all they know is the director. Furthermore, because the behavior is localized in one class, it can be changed or replaced by extending or replacing that class.

Here's how the FontDialogDirector abstraction can be integrated into a class library:
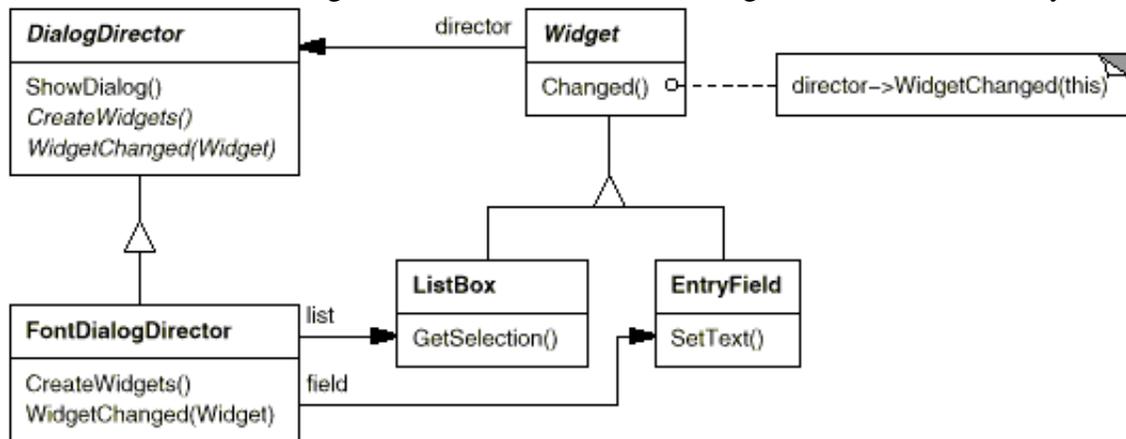

DialogDirector is an abstract class that defines the overall behavior of a dialog. Clients call the ShowDialog operation to display the dialog on the screen. CreateWidgets is an abstract operation for creating the widgets of a dialog. WidgetChanged is another abstract operation; widgets call it to inform their director that they have changed. DialogDirector subclasses override CreateWidgets to create the proper widgets, and they override WidgetChanged to handle the changes.

Applicability

Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways. The resulting interdependencies areunstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.
- a behavior that's distributed between several classes should be customizable without a lot of subclassing.

Structure
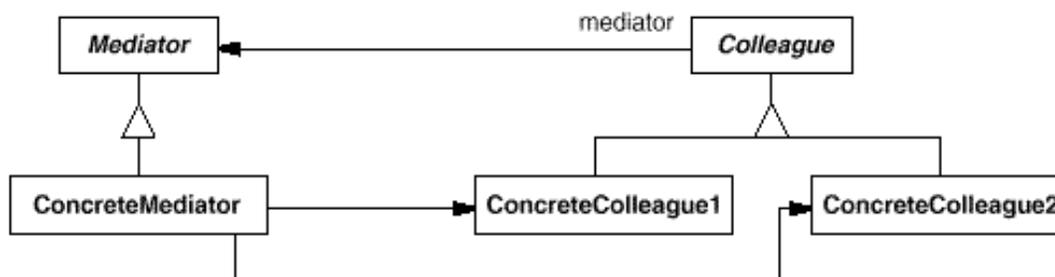
A typical object structure might look like this:
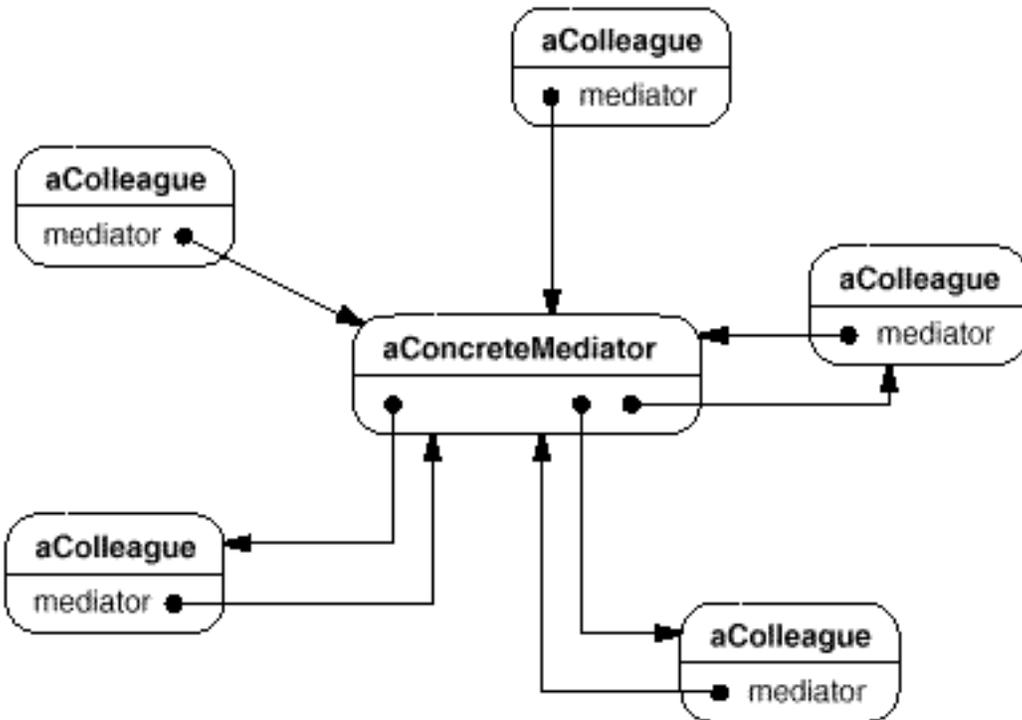
Participants

- Mediator (DialogDirector)
  - defines an interface for communicating with Colleague objects.
- ConcreteMediator (FontDialogDirector)
  - implements cooperative behavior by coordinating Colleague objects.
  - knows and maintains its colleagues.
- Colleague classes (ListBox, EntryField)
  - each Colleague class knows its Mediator object.
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Collaborations

Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

Consequences

The Mediator pattern has the following benefits and drawbacks:

1.It limits subclassing. A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only; Colleague classes can be reused as is.

2.It decouples colleagues. A mediator promotes loose coupling between colleagues. You can vary and reuse Colleague and Mediator classes independently.

3.It simplifies object protocols. A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many relationships are easier to understand, maintain, and extend.

4.It abstracts how objects cooperate. Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior. That can help clarify how objects interact in a system.

5.It centralizes control. The Mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain.

Implementation

The following implementation issues are relevant to the Mediator pattern:

1.Omitting the abstract Mediator class. There's no need to define an abstract Mediator class when colleagues work with only one mediator. The abstract coupling that the Mediator class provides lets colleagues work with different Mediator subclasses, and vice versa.

2.Colleague-Mediator communication. Colleagues have to communicate with their mediator when an event of interest occurs. One approach is to implement the Mediator as an Observer using the Observer pattern. Colleague classes act as Subjects, sending notifications to the mediator whenever they change state. The mediator responds by propagating the effects of the change to other colleagues. Another approach defines a specialized notification interface in Mediator that lets colleagues be more direct in their communication. Smalltalk/V for Windows uses a form of delegation: When communicating with the mediator, a colleague passes itself as an argument, allowing the mediator to identify the sender. The Sample Code uses this approach, and the Smalltalk/V implementation is discussed further in the Known Uses.

Known Uses

Both ET++ [WGM88] and the THINK C class library [Sym93b] use director-like objects in dialogs as mediators between widgets.

The application architecture of Smalltalk/V for Windows is based on a mediator structure. In that environment, an application consists of a Window containing a set of panes. The library contains several predefined Pane objects; examples include TextPane, ListBox, Button, and so on. These panes can be used without subclassing. An application developer only subclasses from ViewManager, a class that's responsible for doing inter-pane coordination. ViewManager is the Mediator, and each pane only knows its view manager, which is considered the "owner" of the pane. Panes don't refer to each other directly.

Smalltalk/V uses an event mechanism for Pane-ViewManager communication. A pane generates an event when it wants to get information from the mediator or when it wants to inform the mediator that something significant happened. An event defines a symbol (e.g., #select) that identifies the event. To handle the event, the view manager registers a method selector with the pane. This selector is the event's handler; it will be invoked whenever the event occurs.

The following code excerpt shows how a ListPane object gets created inside a ViewManager subclass and how ViewManager registers an event handler for the #select event:

```
self addSubpane: (ListPane new paneName: 'myListPane';
owner: self;
when: #select perform: #listSelect:).
```

Another application of the Mediator pattern is in coordinating complex updates. An example is the ChangeManager class mentioned in Observer. ChangeManager mediates between subjects and observers to avoid redundant updates. When an object changes, it notifies the ChangeManager, which in turn coordinates the update by notifying the object's dependents.

A similar application appears in the Unidraw drawing framework and uses a class called CSolver to enforce connectivity constraints between "connectors." Objects in graphical editors can appear to stick to one another in different ways. Connectors are useful in applications that maintain connectivity automatically, like diagram editors and circuit design systems. CSolver is a mediator between connectors. It solves the connectivity constraints and updates the connectors' positions to reflect them.

Related Patterns

Facade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but not vice versa. In contrast, Mediator enables cooperative behavior that colleague objects don't or can't provide, and the protocol is multidirectional.

Colleagues can communicate with the mediator using the Observer pattern.

3. Observer

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As

Dependents, Publish-Subscribe

Motivation

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application

data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.
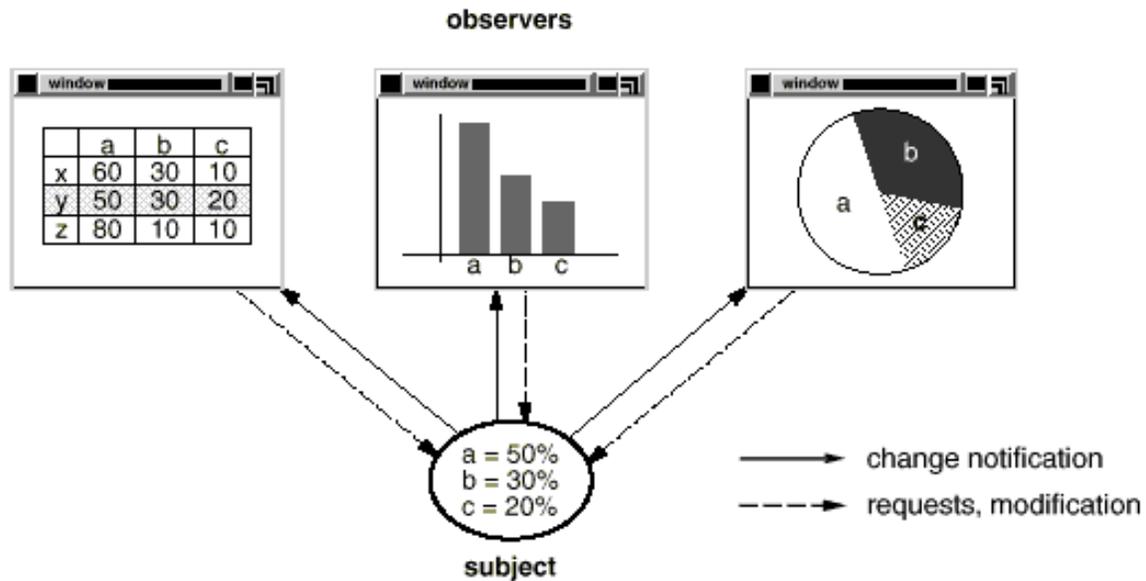
This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. And there's no reason to limit the number of dependent objects to two; there may be any number of different user interfaces to the same data.

The Observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

This kind of interaction is also known as publish-subscribe. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure
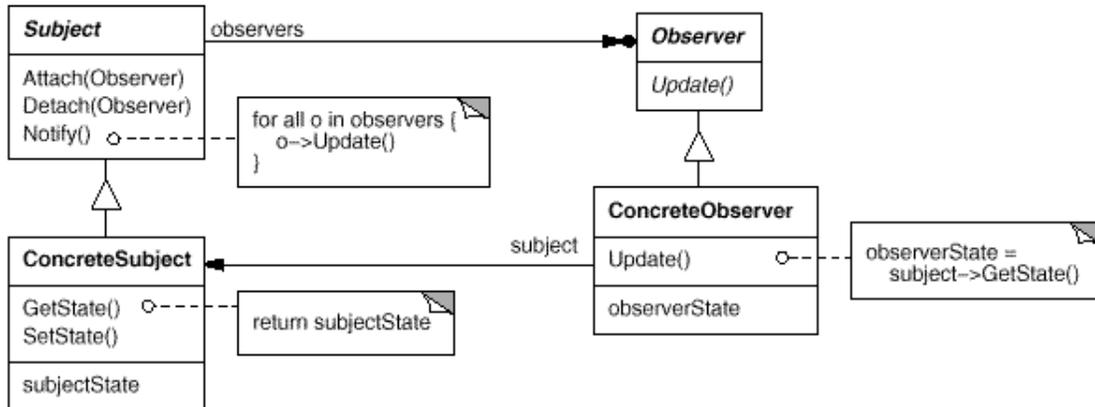
Participants

- Subject
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- Observer
  - defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteSubject
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- ConcreteObserver
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.

After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Consequences

The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following:

1.Abstract coupling between Subject and Observer. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.

Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact. If Subject and Observer are lumped together, then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).

2.Support for broadcast communication. Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.

3.Unexpected updates. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

This problem is aggravated by the fact that the simple update protocol provides no details on what changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

Known Uses

The first and perhaps best-known example of the Observer pattern appears in Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalk environment. MVC's Model class plays the role of Subject, while View is the base class for observers. Smalltalk, ET++, and the THINK class library provide a general dependency mechanism by putting Subject and Observer interfaces in the parent class for all other classes in the system.

Other user interface toolkits that employ this pattern are InterViews, the Andrew Toolkit , and Unidraw. InterViews defines Observer and Observable (for subjects) classes explicitly. Andrew calls them "view" and "data object," respectively. Unidraw splits graphical editor objects into View (for observers) and Subject parts.

Related Patterns

Mediator: By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.

Singleton: The ChangeManager may use the Singleton pattern to make it unique and globally accessible.

B. POAD Development environment user's guide

1. Introduction

POAD Development Environment (PDE) is a CASE tool that supports pattern oriented analysis and design. It facilitates the design and development of software using patterns as basic building blocks. This approach produces designs that can be viewed at different levels of abstraction. At high level of abstraction, the design is viewed as patterns and their dependency relationships. At lower level of abstraction the internals of the patterns can be revealed to produce traditional object oriented class diagrams. In this document, we will discuss the process of creating pattern-oriented designs using this tool.

2. Getting started

2.1 Launching PDE

- If your computer is running Windows 95/98/NT, open the PDE folder and double click on the icon named "run.bat". The PDE will start running.

2.2 Exiting PDE

To exit PDE, select "exit" from the "File" menu in the main window.

3. Working with design models

In the PDE, a design is represented by a design model. A design model can contain multiple diagrams that provides different views of the design.

3.1 Creating a new design model

To create a new design model, open the "File" menu in the main window and select "new". A blank design model will be created.

3.2 Opening an existing design model

To open an existing design model, open the "File" menu in the main window and select "open …". A file dialog will appear that lets you select a file. Click on the "Ok" button in the file dialog. If the selected file is of valid format, it will be opened and displayed as a design model.

3.3 Saving a design model

To save a design model, select "Save" or "Save as" from the "File" menu. "Save" will cause the design model to be saved under current name. If "Save as" is selected, a dialog window will appear that lets you enter a name for the design model. Click on the "Ok" button in the dialog window, the design model will be saved under that name. The action of "Save" on a untitled design model will have the save effect as "Save as".

4. Working with diagrams

There are three levels of diagrams supported by PDE: pattern level diagram, pattern interface diagram and detailed pattern diagram. Each level of diagram provides a different view of the pattern or patterns involved.

4.1 Pattern level diagrams

Creation

To create a pattern level diagram,
- Select "create" from the "Diagrams" menu
- Click right mouse button over the "Pattern Level View" folder in the Browser window and select "create" from the popup menu

A dialog window will appear that prompts you for the name of the diagram. Enter the name and click on "Ok". A blank pattern level diagram will be created and appear inside the PDE main window.

Editing

- Adding patterns   Select the "Pattern" drawing tool in the working toolbar and then click on the diagram where you want the pattern to be added. A pattern symbol will be added to the diagram.
- Adding subsystems   Select the "Subsystem" drawing tool in the working toolbar and then click on the diagram where you want the subsystem to be added. A subsystem symbol will be added to the diagram.
- Adding dependency relations   Select the "Dependency" drawing tool in the working toolbar, then press the left mouse button on a pattern/subsystem and drag the mouse to another pattern/subsystem. Release the mouse button, a dependency relation will be added to the pattern/subsystem pair.
- Adding note   Select the "Note" drawing tool in the working toolbar and then click on the diagram where you want the note to be added. A note symbol will be added to the diagram. Click on the note and you will be able to enter text.
- Adding text   Select the "Text" drawing tool in the working toolbar and then click on the diagram where you want to enter text. A blinking cursor will appear and you will be able to enter text.
- Anchoring note to an itemSelect the "Note connection" drawing tool in the working toolbar, then press left mouse button on a note and drag the mouse cursor to an item. Release mouse button and a note connection will be added between the note and the item.
- Selecting and moving items   Select "Select" tool in the working toolbar, you will be able to select and drag any symbols in the diagram.
- Editing specification of items   Select an item and click on the right mouse button, a popup menu will appear that lets you edit the specification of the item.

4.2 Pattern Interface Diagrams

Creation

To create a pattern interface diagram, double click on a dependency relation symbol in a pattern level diagram. If the dependency relation has a name, a new pattern interface diagram with that name will appear. Otherwise, a dialog window will appear that asks you for the diagram name. Enter the name and click on "Ok", a new Pattern Interface diagram with that name will be created.

Editing

- Adding items to pattern interface Click right mouse button on a pattern symbol, a popup menu will appear that lets you add a class or a method to the pattern interface.
- Adding association relation between interface items Select "Link" drawing tool in the working toolbar, then press left mouse button on an interface item and drag the mouse to an interface item of another pattern. Release the mouse and an association relation between the two items will be added.
- Resizing patterns Select a pattern, four small rectangles will appear at four corners of the pattern symbol. Use the bottom right rectangle as a handle to resize the pattern.
- Adding note Select the "Note" drawing tool in the working toolbar and then click on the diagram where you want the note to be added. A note symbol will be added to the diagram. Click on the note and you will be able to enter text.
- Adding text Select the "Text" drawing tool in the working toolbar and then click on the diagram where you want to enter text. A blinking cursor will appear and you will be able to enter text.
- Anchoring note to an itemSelect the "Note connection" drawing tool in the working toolbar, then press left mouse button on a note and drag the mouse cursor to an item. Release mouse button and a note connection will be added between the note and the item.
- Selecting and moving items Select "Select" tool in the working toolbar, you will be able to select and drag any symbols in the diagram.
- Editing specification of items Select an item and click on the right mouse button, a popup menu will appear that lets you edit the specification of the item.

4.3 Detailed Pattern Diagram

Creation

Double click on a pattern symbol in a Pattern Level diagram or a Pattern Interface diagram. A dialog window will appear which prompts you for the name of the diagram. Enter the name and click "Ok" button. A new detailed pattern diagram will be created.

Editing

- Adding items to pattern Click right mouse button on a pattern symbol, a popup menu will appear that lets you add a class to the pattern internal structure.
- Resizing patterns Select a pattern, four small rectangles will appear at four corners of the pattern symbol. Use the bottom right rectangle as a handle to resize the pattern.
- Adding note Select the "Note" drawing tool in the working toolbar and then click on the diagram where you want the note to be added. A note symbol will be added to the diagram. Click on the note and you will be able to enter text.

- Adding text   Select the "Text" drawing tool in the working toolbar and then click on the diagram where you want to enter text. A blinking cursor will appear and you will be able to enter text.
- Anchoring note to an itemSelect the "Note connection" drawing tool in the working toolbar, then press left mouse button on a note and drag the mouse cursor to an item. Release mouse button and a note connection will be added between the note and the item.
- Selecting and moving items   Select "Select" tool in the working toolbar, you will be able to select and drag any symbols in the diagram.
- Editing specification of items   Select an item and click on the right mouse button, a popup menu will appear that lets you edit the specification of the item.

C. Pattern Browser User's Guide

1. Introduction

The Pattern Browser is a Java applet that enables a user to browser the content of our Design Patterns Database. It can run inside any web browser that supports Java Swing. The database is located on a remote server which the applet make connection to using JDBC. The Pattern Browser lists the names of all patterns available in the database. The user can select the pattern he/she is interested in and view its details.

2. Getting started

To start the Pattern Browser, just load the page that contains it. The Pattern Browser will automatically start running. The following picture shows its user interface.
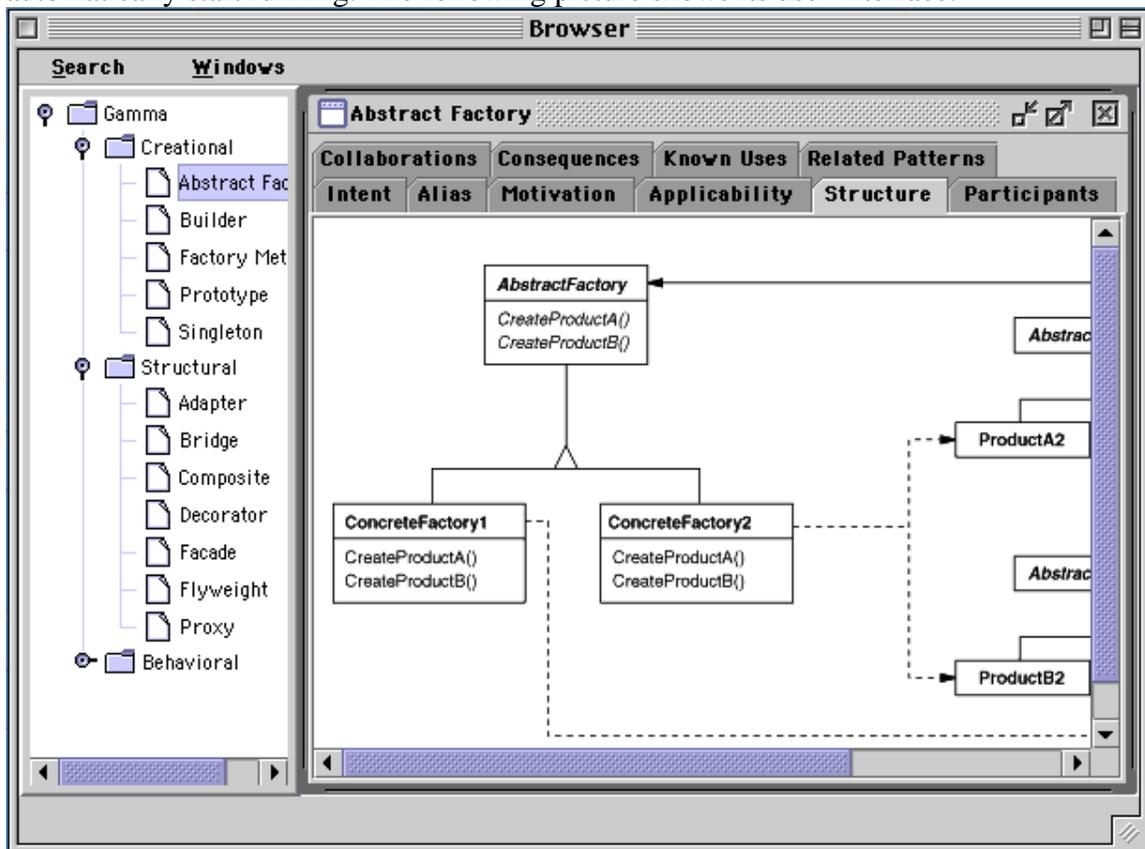
3. Viewing pattern directory listing

The design patterns in the database are organized into pre-defined categories. A category is like a folder, it can contain patterns or other categories. The Pattern Browser displays this hierarchy of categories and patterns using a tree like structure. A category is represented using a folder icon and a pattern is represented using a file icon.

- To expand a category, double click on the folder icon of the category. The category will be expanded and names of all its patterns will be displayed.

- To view the detail of a pattern, double click on the file icon of the pattern. A new window will be opened inside the applet. The details of the pattern will be displayed in the window.

4. View details of a pattern

The details of a pattern, including its name, intent, motivation, structure, etc. are displayed inside a window with tabbed panels. Each panel contains an element of the documentation for the pattern. For example, the "Motivation" panel will contain the text that describe the motivation for this pattern. You can click on any tab to view certain information about the pattern. In "Related Patterns" panel, the text may contain some hyperlinks that points to other patterns. You can jump to these patterns by clicking on the hyperlink.