1999

Integration of Chicory components and Chicory optimization

Avinash Venkatesh Kalgi
West Virginia University

Integration of Chicory Components And Chicory Optimization.

By

Avinash V. Kalgi.

A Thesis

Submitted to
The College of Engineering and Mineral Resources
at
West Virginia University

in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Department of Computer Science and Electrical Engineering
West Virginia University
Morgantown, West Virginia
1998

Integration of Chicory Components and Chicory Optimization.

Avinash V. Kalgi.

(Abstract)

Software is not built as monolithic structure. It is built in blocks by more than one person. This software then has to be put together and made to work. It is also important to ensure that the assembled software is performing optimally.

Chicory™ is such a Java™ software. It is made of numerous components, made by a lot of different people.

This thesis explores the complications associated with integrating these components. This is achieved by an exhaustive description of the architecture of the components and a detailed description of the design decisions. It explains in detail the interactions between various objects inside Chicory™. To explain the structure we first give an overview of the system and then explain the structural details and follow it by significant object interactions.

We also take care to explain the steps to be followed when extending the software to add functionality.

Software when built is not initially in its most optimized form. Structures and control flows exist which slow the application down when exposed to heavy loads. Data types used may not be fast enough to allow at least usable performances. Computation might be unnecessarily repeated. This thesis also explains the methods that we followed to optimize Chicory™. We explain methods applied to make Chicory™ use less memory and run faster and eliminate the problems explained above.

In putting forth these explanations we hope to impress on the user, the complexity associate with managing software of large proportions. We hope that the reader will gain significant insight into the functioning of Chicory TM.

Acknowledgments

I would like to express my sincere thanks to my advisor Dr. Srinivas Kankanahalli for all the support that he extended to me. I am extremely grateful to him for granting me the freedom to explore and learn. I am also thankful to my committee members Dr. Jagannathan and Dr. Sumitra Reddy for their help and guidance.

I would also like to thank the people from whom I learnt a lot namely Dale Moore and Jeremy Sigmon.

A special thanks to my friend Ravi Banda for sharing the work on the Chicory project.

Table of Contents.

List of Figures.

Chapter1: Introduction

1.1 The Chicory project

Chicory[25] is an Integrated Software Application Development Environment. It supports the entire software application development life cycle [26].

The traditional software life cycle encompasses requirement elicitation, specification, analysis, design, coding, automatic test generation, maintenance and reengineering support. Chicory not only supports traditional software life cycle, but also supports application development by providing tools for creating, populating and querying databases and Web servers. Chicory is a language-based (in this case the specific language is Java [15]) environment, which means that various tools which constitute the Chicory environment are Java aware.

Chicory is a visual application development environment for rapidly creating client-server Java applications. It enhances programming productivity by insulating the application developer from unnecessary programming details and automatically generating the Java code. It saves software developers' time in the designing phase, coding phase and testing phase.

Chicory currently supports the construction of three-tier client-server architectures using the Java Database Connectivity (JDBC) mechanism [1]. Chicory has a visual database client and also supports generation of Database forms. Facility to generate reports based on user's choice of columns and formats has also been added. Chicory supports component assembly and component-level software reuse by supporting the Java Beans [23] component framework. Chicory supports user-defined components to be imported into the Chicory framework and its palette. Those features allow application

developers not only to use third-party components in their applications but also to use the visual metaphors provided by Chicory to interact with their own components.

One of the exciting additions to Chicory has been the incorporation of the UML facility. This facility provides the user with a tool to design a software application using Class, State, Interaction and Usecase diagrams. The tool also allows for code generation with Java being the target language.

In the near future, Chicory will support distributed interoperable architectures such as the Common Object Request Broker Architecture (CORBA) and the Distributed Component Object Model (DCOM), which will provide middleware services and ensure architectural scalability.

Chicory includes for example tools for visual graphic user interface builder, visual debugging, cross-referencing tools, editors, object browsers, profilers, static analyzers and lightweight program databases.

The Chicory project was initiated and named by Dr. Srinivas Kankanahalli at Concurrent Engineering Research Center (CERC), at West Virginia University.

1.2 The reason for choosing Java [15].

Any software is written in at least one kind of programming language. Looking at the history of programming languages, the earliest programming paradigms were chaos programming such as BASIC with “jump” and “go to” sentences anywhere. The functional programming was a major improvement over the chaos programming due to the code reuse idea. The structured programming which came after functional programming provided the neat loops and “if .. then .. else” structure which reduced the

path and made debugging easier. Finally came the object-oriented paradigm. The features of object-oriented programming include data abstraction, encapsulation, inherence and polymorphism. Object-oriented languages are Smalltalk, C++ and Java, among others.

One of the fundamental differences between C++ and Java is that Java is platform-independent. A Java program written on PC, can run well on UNIX, or Mac without any changes. Java is simple and easy to learn. Java does not have pointers. Java automatically does garbage collection to delete objects that are no longer in use. The programmer does not need to remember to free up memory or to destroy objects. However, pointer manipulation, memory allocation and deallocation can easily cause problems in C++ language.

Java is designed to support applications on networks and is a distributed language . Remote Method Invocation (RMI) enables the programmer to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. Java is secure, it protects you from untrusted applets. JDBC is a standard SQL database access interface, providing uniform access to a wide range of relational databases. One of the best things about Java is its Java Beans APIs. The goal of the Java Beans APIs is to define a software component model for Java, so that third party ISV's can create and ship Java components that can be composed together into applications by end users.

1.3 Recent Changes:

Recently Javasoft introduced JFC [24] [22] i.e. Java foundation classes. JFC is a class library that is designed to help developers to build full-featured enterprise-ready applications.

JFC contains

a) The Swing toolset: A comprehensive set of *lightweight* components with a pluggable look-and-feel design.

b) Java 2D: A set of classes for advanced 2D graphics and imaging. Java 2D supplies Java applications with many different paint styles, mechanisms for defining complex shapes, and classes and methods for fine-tuning the rendering process.

c) Drag and Drop: a technology that enables data transfer across Java and native applications, across Java applications, and within a single Java application.

d) Accessibility API: An interface that provides assistive technologies such as screen magnifiers and audible text readers. These technologies are designed to help people with disabilities interact and communicate more easily and efficiently with JFC and AWT components.

It was decided in January 1998 that Chicory should leverage the JFC technology. This provided the benefit of incorporating pre existing Swing components into the structure of Chicory thus bringing down the cost of Code maintenance. It also released the developer from designing GUI components and allowed him to concentrate on the business logic. As Chicory stands today it is 100% Java.

The next chapter outlines the basic structure of Chicory™ . It also explains the inner workings of the startup of Chicory™

Chapter 2 : Structure of the Application

2.1 Introduction

The aim of this chapter is to explain the complex structure of Chicory. The explanation is carried out using Class diagrams, Interaction diagrams and textual explanation as well [4]. Initially the basic elements of the application are explained after which each component is discussed in detail. Please note that diagrams are intended to give an *idea* of the structure and processes in Chicory. The diagrams make a reasonable attempt to depict the *variable-by-variable* structure and the *method call -by- method call* sequence.

2.2 Startup

The Starting file for Chicory is *Chicory.class*. This class file is situated in a jar [27] called as Chicory.jar. This jar file is the starting point for the application. To run Chicory the command *java -jar Chicory.jar* is invoked.

The jar file has another jar file called *trans.jar* identified in the Classpath variable of it's manifest. The 'trans.jar' is situated in the directory called classes. The Classpath variable in the manifest of *Chicory.jar* is hence *classes/trans.jar*. Shown below is a sequence of events during startup.

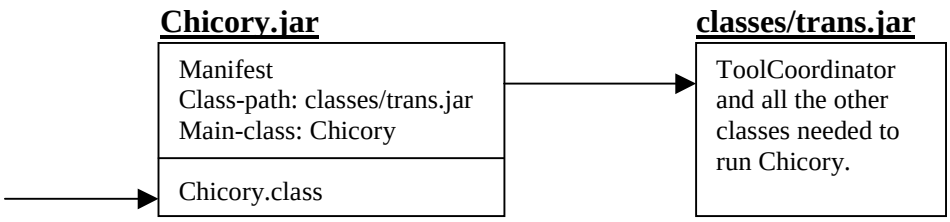


Figure 1: The startup structure.

Here is what happens :

a) The Command 'java -jar Chicory.jar' is used to run the application. This command looks in the manifest of Chicory.jar for the class to run. The class that is found is Chicory.class.

b) Chicory.class is run by calling it's constructor. The constructor has code to load a file called as ToolCoordinator and create a new instance of the loaded class. Java searches for ToolCoordinator.class in the class-path variable of the manifest of Chicory.jar. The class-path variable points to trans.jar in the classes directory. ToolCoordinator.class is loaded from trans.jar and an instance is created. The ToolCoordinator is the coordinator class for all the activities of Chicory and is explained in the next section.

2.3 The ToolCoordinator:

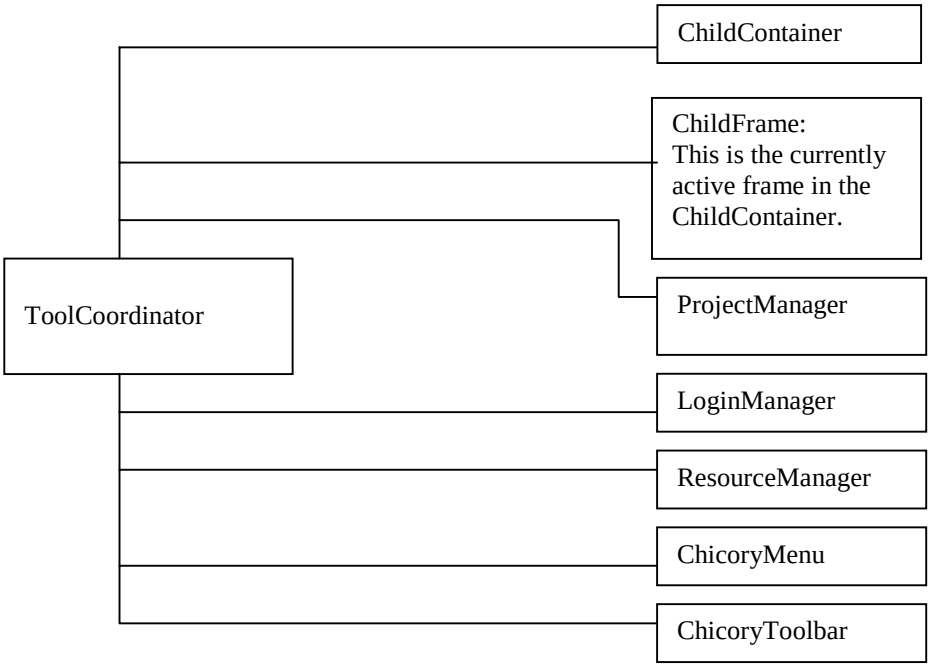


Figure 2: The basic structure.

Shown above are the main constituent classes of the Toolcoordinator. The Toolcoordinator is the coordinating class for Chicory. It is the top level frame and is thus the parent for all chicory components. In addition it performs the task of listening to the trees of in the ProjectManager.

2.4 The ChildContainer [10]:

This class is the MDI (multiple document interface) of Chicory. ChildContainer extends a JDesktopPane [22] and can hold ChildFrame(s) (explained later). The ChildContainer also assumes the responsibility of adding these ChildFrames via it's *'addChild(ChicoryTool) '* method. One other important task performed by the ChildContainer is the *'finding'* of particular ChicoryTool. This is done with via it's multiple *'findXXX()'* methods. Whenever a particular ChicoryTool needs to be found, the

appropriate *'findXXX()'* method is called with the ChicoryTool as the parameter. The ChildContainer then goes through all the existing frames and brings the frame containing the tool to the fore. If the tool is not currently present in the ChildContainer then a ChildFrame is instantiated and the tool put inside it. Here is an explanatory diagram.

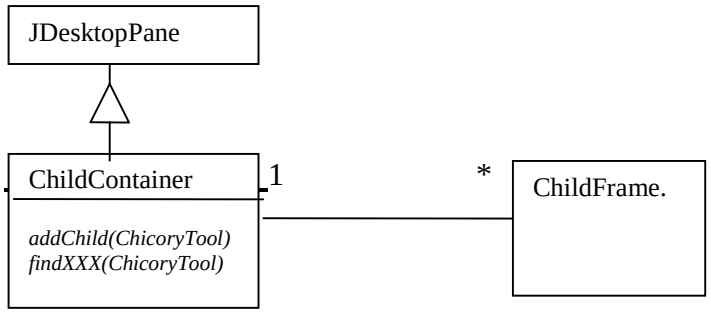


Figure 3: The ChildContainer Structure.

2.5 The ChildFrame [10]:

The ChildFrame extends the JInternalFrame and contains a ChicoryTool. The ChicoryTool holds the program that wants to run in the ChildFrame (and hence in the ChildContainer). The ChildFrame gets it's title from the ChicoryTool that it is holding. One of the main functions of the ChildFrame is the initiation of the method calls to update the tool palette status. Here is how it is performed. Some of the tools inside the ChildFrame require Java beans [23] [2] to operate. Typically these kind of tools are designers (ex. GUIDesign [25], UMLDesigners, ERDiagrammer etc…). One of the concerns while using these tools is the availability of beans relevant *only* to the designer that has focus. This is achieved by the ChildFrame calling the *'requestFocus()'* method of the contained ChicoryTool when the ChildFrame is selected. The ChicoryTool then makes calls to update the status of the tool palette and thus display the beans that are relevant to the designer. Here is a diagram that shows the structure and the process

explained above.

Figure 4: The ChildFrame Structure.

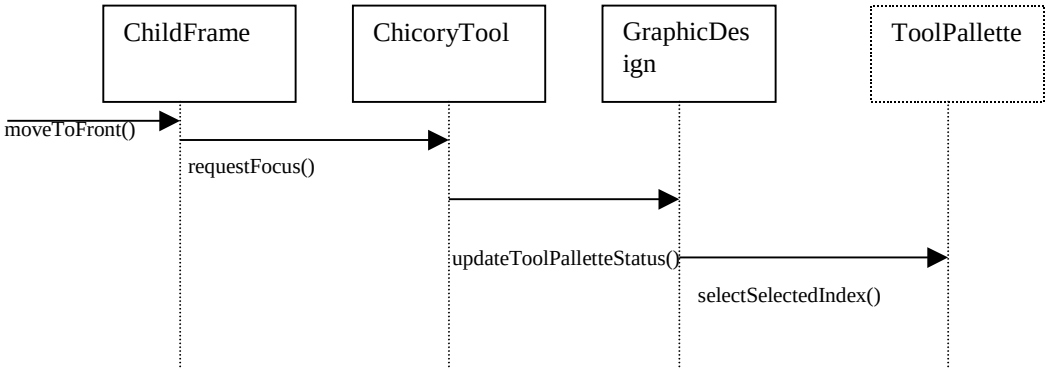


Figure 5:Interaction diagram for the tool palette update.

2.6 The ProjectManager:

The project manager is responsible for handling the project related functions of Chicory. These functions include holding references to the Project name & directory. Objects like FileManager, Debugger [3] and the tabbed pane for the trees (database, components, Uml, files, help) reside inside the project manager. The Project Manager also implements the DebuggerCallBack interface. This interface allows the Debugger to call back the Project Manager when it performs a operation like hitting a break point. The Project

Manager also holds a reference to a Tester object. This Tester object is used to handle the functionality of testing an Applet or an Application. The 'Tester' is basically an interface, which is implemented by the 'ApplicationTester' and the 'AppletTester' classes.

Here is a diagram showing the associations of the ProjectManager with the constituent components.

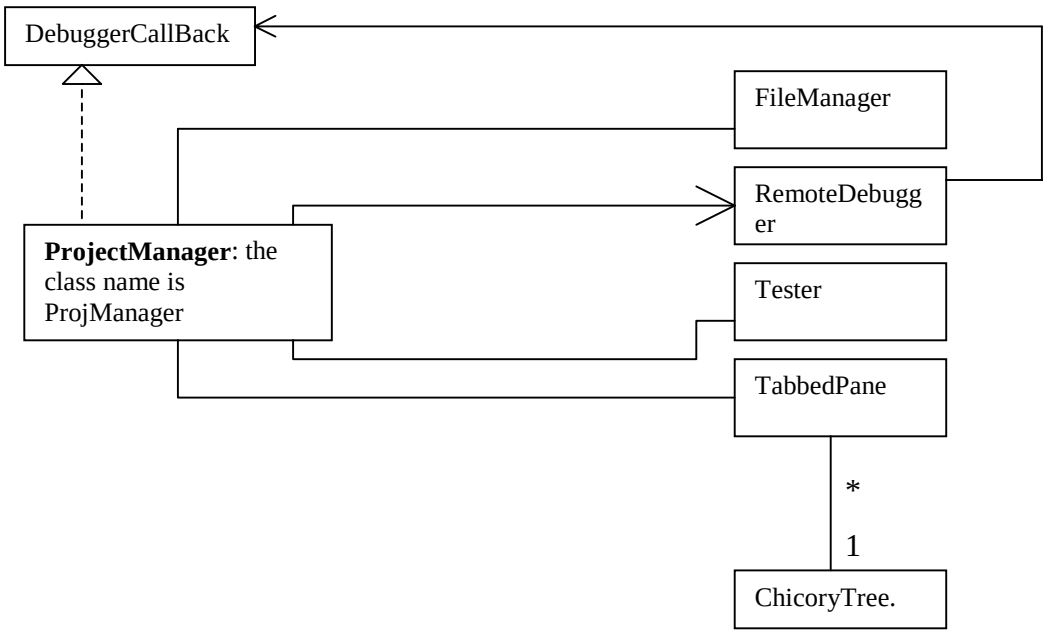


Figure 6: The ProjectManager.

2.7 LoginManager:

The LoginManager is the base of all the database [1] functionality of chicory. The LoginManager has it's own tree which it uses to display the various components that it contains. The main component that it contains is the list of DatabaseManager objects. The DatabaseManager further contain all the rest of the database functionality i.e. Query, Update, Insert, Delete [21] operations and Database Reports functionality. The LoginManager is further responsible for being first link in the process of serializing and de-serializing a database client. The Diagram below shows the Class structures and the interaction diagram for the serialization actions.

Here is a brief description of the Serialization process.

a) The LoginManager is asked to return the State of the Database client. The LoginManager creates a LoginManagerState object and asked it to gather all the state information. The state object goes through it's list of LoginManager's DatabaseManagers and asks each one to return it's state.
b) Each of the Database managers then creates DatabaseManagerState object and asks it to gather the state for the DatabaseManager. The state object in turn asks the each of the sub managers (query, update…) to return state. These return an OperationManager state object . The DatabaseManager then adds the table information and login information and returns the state object.
c) The LoginManagerState adds the DatabaseManagerState object to a list and on gathering all the states is returned to the serializing process.

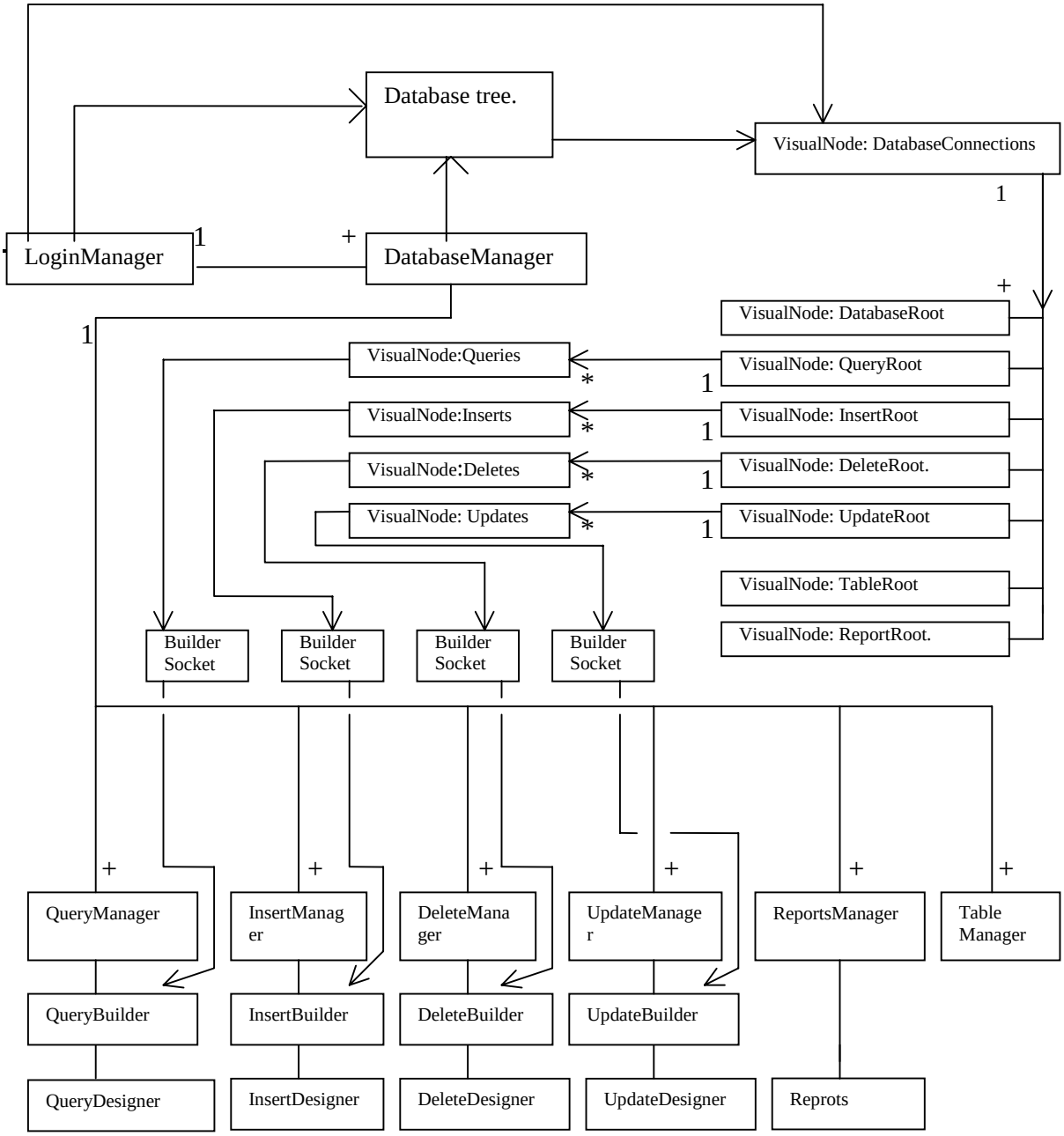


Figure 7

Database functionality Class Diagram.

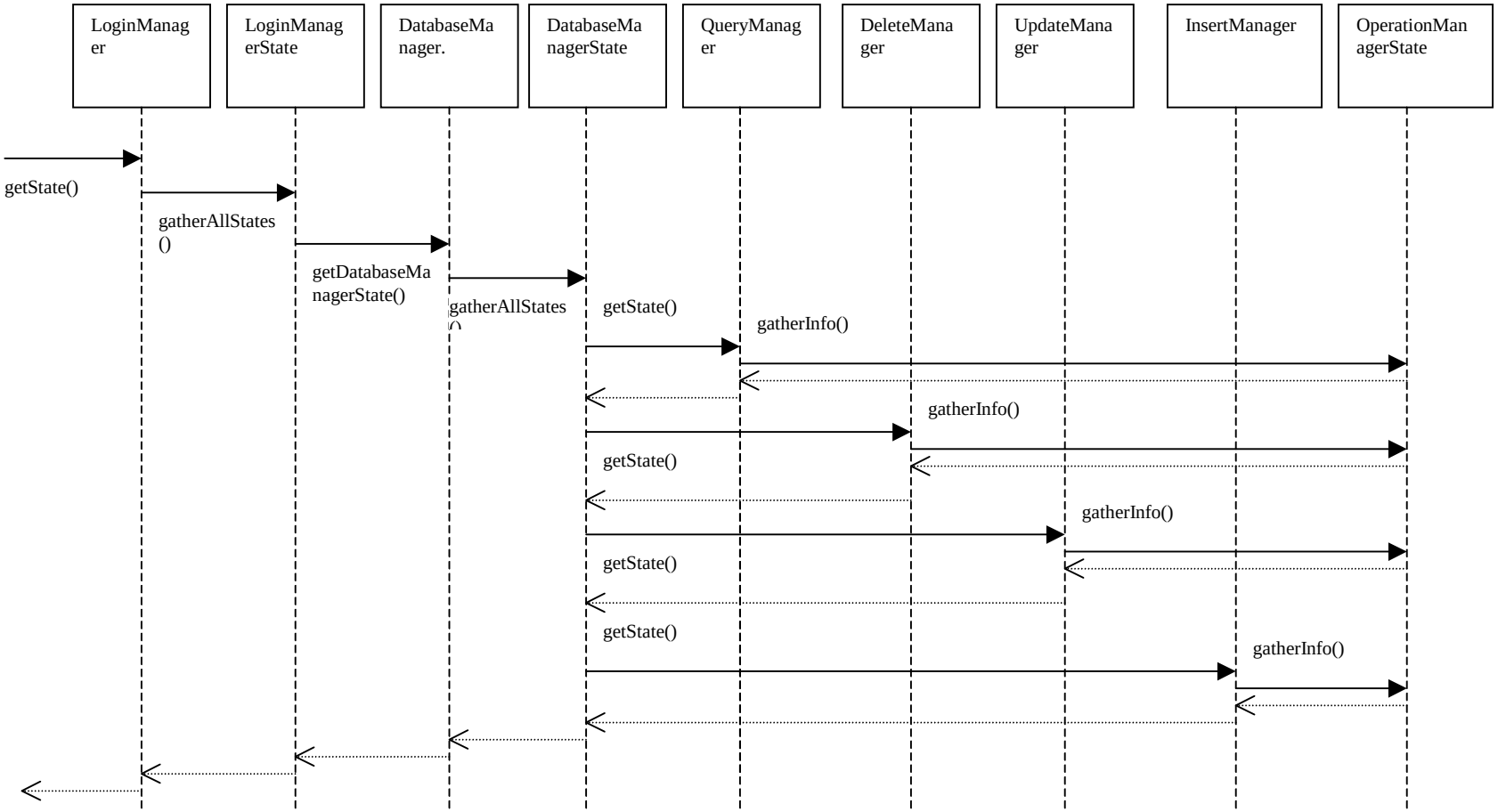


Figure : 8.

Command sequence for the Serialization process of the Database client.

2.8 The ResourceManager:

The ResourceManager is the place where all the commonly required functionality of Chicory is stored. These include

1) JFileChooser [11] [22] required for all the file access function in Chicory.

2) The PrinterManager [13] used for printing purposes.

3) The ApplicationTester and the AppletTester.

4) The ClasspathSetter used to handle all Classpath setting & references.

5) The HelpBrowser and the HelpContents. These form the Chicory help structure.

2.9 ChicoryMenu:

The ChicoryMenu is the '*point and click'* initiator of processes inside Chicory.
The ChicoryMenu listens to it's own menu items and instantiaes commands for the ToolCoordinator to execute (the command pattern approach). There are some occasions when the command instantiated is valid only in the context of the currently focused ChicoryTool in the ChildContainer. In such cases, ChicoryMenu passes on the action event to the ProjectManager. It then becomes the ProjectManager's responsibility to rout the command to the focused tool (see the *'actionPerformed'* method in the code of the file

ProjManager.java. The following diagram shows the relation between the ChicoryMenu and it's menu items.

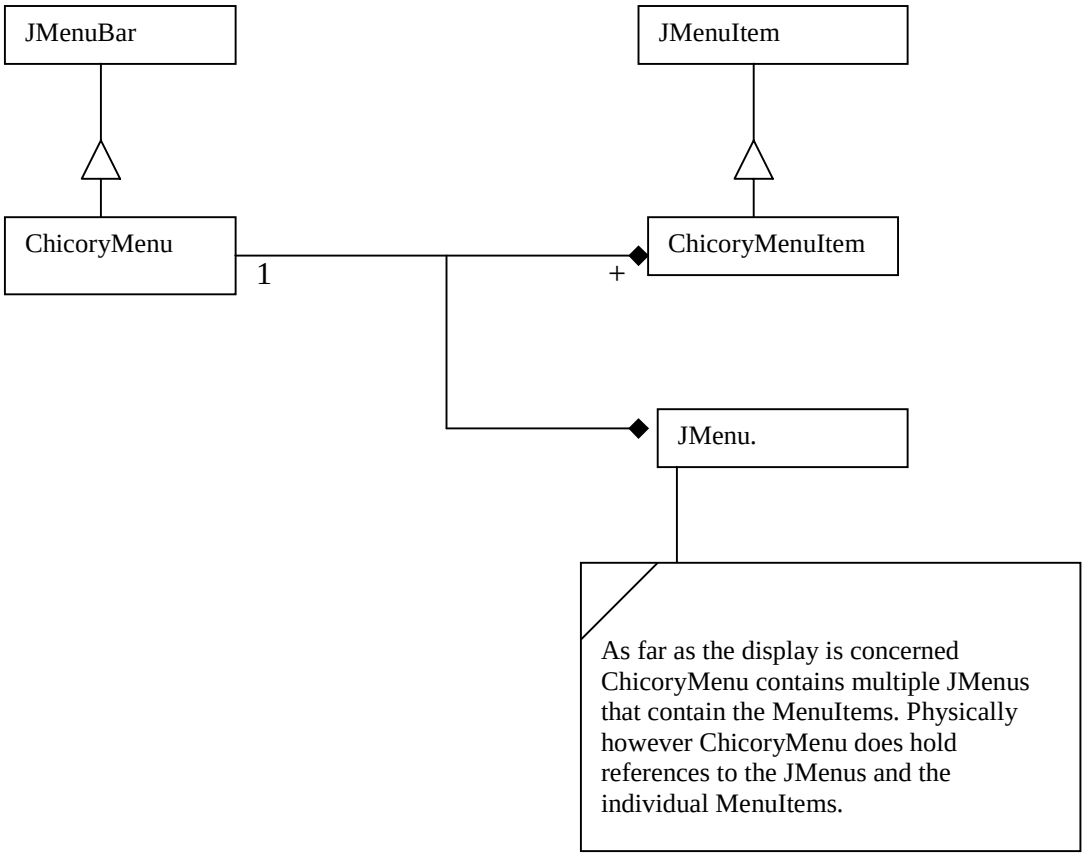


Figure: 9: The ChicoryMenu structure.

2.10 ChicoryToolBar:

The ChicoryToolBar contains buttons for easily accessing the commands without going through the ChicoryMenu. The ToolBar contains ChicoryButtons and a textfield (editor text search functionality). The ChicoryButtons are instantiated with knowledge of their related ChicoryMenuItems. The buttons are also their own ActionListeners. Here is what happens when a button is clicked.

a) The ChicoryButton listens to it's own action.

b) It then creates an action event with the related ChicoryMenuItem as the source.

c) The *'actionPerformed'* method of the ChicoryMenu is called with the instantiated ActionEvent. The rest is as explained for the ChicoryMenuItems.

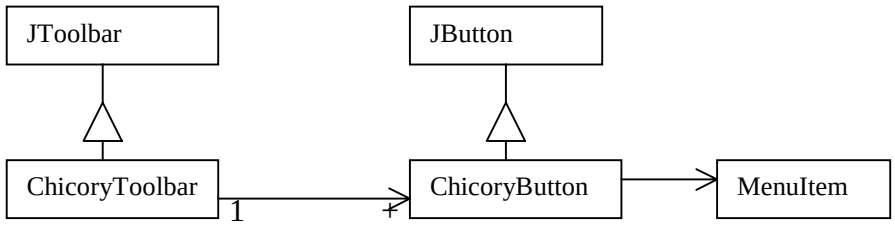


Figure: 10: The ChicoryToolbar structure

Chapter 3 :The Text Editor

3.1 Introduction:

This chapter explains the working of the Text editor of Chicory. The editor leverages the Text framework of Java 1.2 [8] [9]. The editor provides syntax coloring for source code written in Java and also provides support for coloring single/multi-line comments. The editor also provides an API to perform the normal editing tasks of cut, paste, copy, undo/Redo and find. Other functionality include the setting and clearing of breakpoint. The following section begins the explanation of the editor.

3.2 Structure:

Before delving to the structure the reader is advised to be familiar with the text framework of Java 1.2. This section does not explain the framework. One source of information about the text framework is the site [8] [9].

ChicEditor.class is the main class for the editor functionality. It extends the JEditorPane [22]. ChicEditor depends mainly on the following classes for functionality.

a) ChicoryDocument.class: extends PlainDocument [22] and adds comment recognition code to the structural storage aspect of PlainDocument [22].

b) ChicoryContext.class: A collection of styles used to render Java text. This class also acts as a factory for the views used to represent the Java documents. This class extends StyleContext(Java API) and implements ViewFactory [22] interface.

c) ChicoryEditorKit.class: This class extends the DefaultEditorKit [22] and implements the CommentListener interface. The class performs the function of an editor kit and also provide a means of updating the view once a change in the comment status occurs.

d) Token.class: This class provides a convenient to access Java tokens lexical token. This wraps the Constants used by a Scanner (Java API) to provide a convenient class that can be stored as a attribute value.

The Following class diagrams depict the structure of the editor.

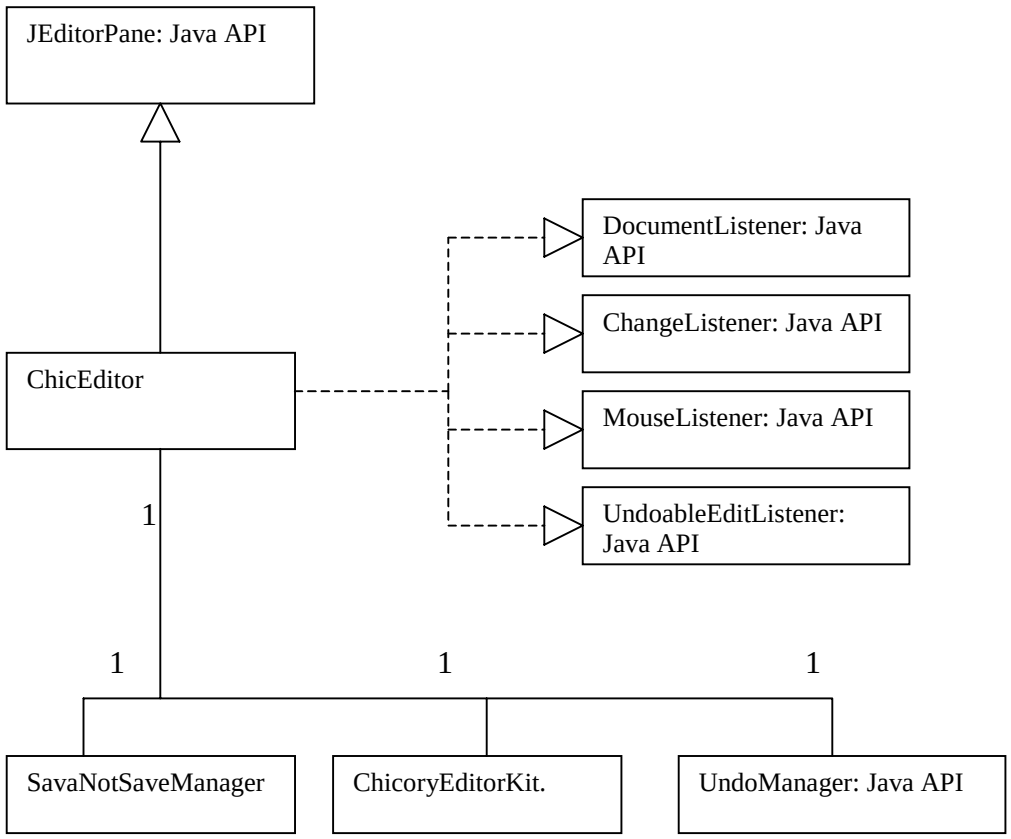


Figure11: The ChicEditor.

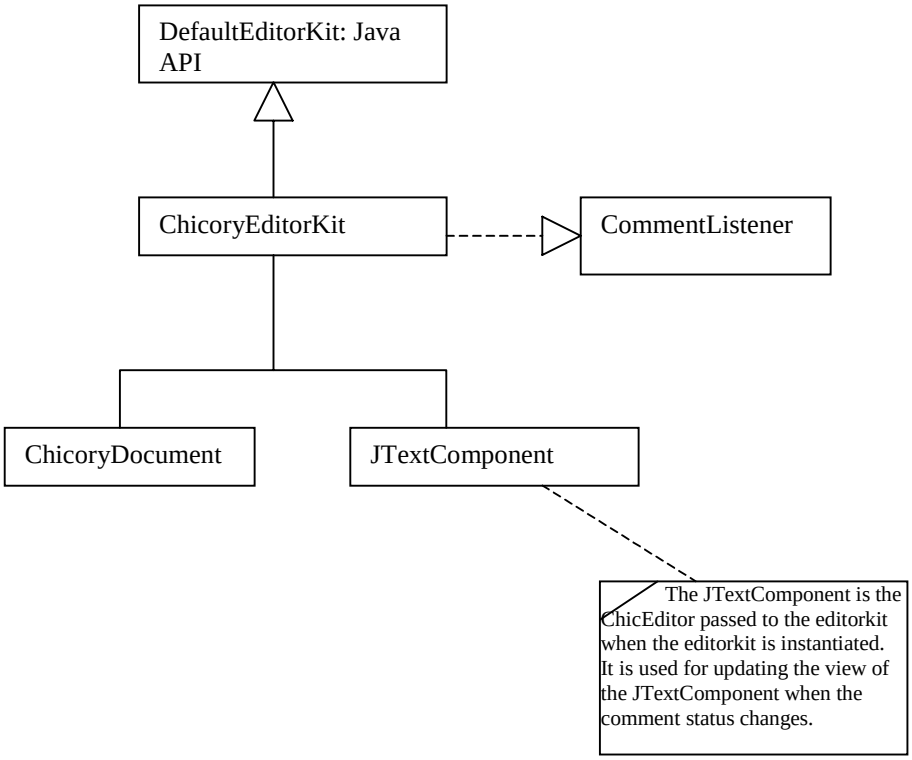


Figure12: The ChicoryEditorKit.

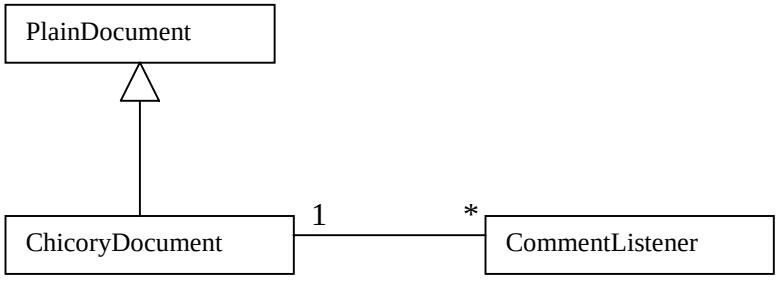


Figure13: The ChicoryDocument.

3.3 Explanation of the logic:

This section explains the basic logic on which the syntax coloring of the editor works. Since the text editor functionality leverages the text framework of Java 1.2 [22], the structure follows a variant of the MVC architecture. The following explanation proceeds in the context of that architecture.

a) The ChicoryDocument forms the model for the ChicEditor. The ChicoryEditorKit is the EditorKit for the ChicEditor. It instantiates and returns the ChicoryDocument and ChicoryContext when it is set as the EditorKit for ChicEditor. The Chicory context implements the ViewFactory interface and hence is responsible for generating and returning a *'View'* [22] for a given *'Element'* [22]. The *'View'* returned is ChicoryView (inner class of ChicoryContext). ChicoryView extends WrappedPlainView [22]. ChicoryView is responsible for rendering the text for the element that it is created for. This means that ChicoryView needs to know if the element that it displays is a part of Java code or comment. Based on this information it can either syntax color the text or paint it in the color allocated for comments.

b) When text is typed into the text editor it first gets added to ChicoryDocument. The *'insertUpdate'* method of the Document is called to actually add the text. The *'insertUpdate'* method of PlainDocument[22] is over written in ChicoryDocument to add the comment recognition code. The PlainDocument's *'insertUpdate'* is called first so that the Document structure is updated. Following this call we check to see for the presence of "/*" or "*/" string to ascertain if a comment is started of ended. If we detect a "/*" we know that a comment was started hence this line is labeled as a Comment. This is done by adding the *'CommentAttribute'* object to the AttributeSet

of the line. *'CommentAttribute'* is an object of type *'AttributeKey*'. The *'AttributeKey*' class is a static inner class of ChicoryContext and basically exists to perform the function of a marker. When a line has this class in it's *'AttributeSet'*, the line is considered as a comment. Various combination of checking for the presence/absence of the *'CommentAttribute'*, "/*" and "*/" are performed to ascertain if the line is a Comment. Similarly when a deletion occurs in the text editor the PlainDocument's *'removeUpdate'* method is called to effect the change. This method has been over written in the ChicoryDocument to update, if any, changes in the Comment status. The PlainDocument's *'removeUpdate'* is first called to update the Document structure after which the code for updating the Comment status is included. Changing the Comment status involves adding/removing the *CommentAttribute* from lines added /removed from a Comment when the string deleted is a "/*" or a "*/".

c) Following the insert/deletion of text, the appropriate *'View'* is updated. Note that view used here is the ChicoryView. The *'drawUnselectedText'* method of this class is overwritten to display the allocated text properly. It must be kept in mind that each *'View'* is allocated a line to display. The line allocated is indicated by the start position and end position of the line in the ChicoryDocument. Here is what typically happens. When the *'drawUnselectedText'* method is called, the Element corresponding to the line allocated to display is obtained. This Element is checked for the presence of the *'CommentAttribute'*. For the purposes of keeping this explanation simple we assume that the line contain or does not contain a *'CommentAttribute'*. If the line contains a *'CommentAttribute'* then we paint the complete line using the color for the comment. If the line does not contain the *'CommentAttribute*, we send it to a method called as

'paintSyntaxColoredText' this method sends the line to the Scanner (updates the scanner to the range of the line). The scan of the method is then called repeatedly to get the tokens. When the token is obtained we paint it to the *'View'* using the appropriate color. The color is obtained by a call to the *'getForeground*' method. This method takes the token as the only argument and returns the Color for the token. This explanation provided the simplest scenario. Conditions exist where code and comments can exist on the same line.

d) When the Comment Status of the document changes, the ChicEditor needs to reflect the total change i.e. when a "/*" is added to a line the text after the "/*' becomes a comment. Now if the comment is not ended the comment continues till it meets a "*/". What would typically happen is that only the *'View'* related to the changed would be updated, hence the current line would be indicated as a comment but the change would not ripple to the subsequent lines. To facilitate this change the Concept of a CommentListener was created. The ChicoryEditorKit implements the CommentListener and registers itself as a CommentListener with the ChicoryDocument. When a change in the Comment Status occurs, ChicoryDocument notifies the CommentListeners. Each CommentListener then tells the related ChicEditor (JTextComponent) the range of the subsequent document to repaint so that the Comment Status change can get reflected.

e) Here is typically what happens when we type in text.

   1) The *'insertUpdate'* method of the document is called with the additions.
   2) The additions are made to the document are made and the Comment status is updated.

3) The *'View'* is then called to reflect the additions.

3.4 Other functions:

BreakPoint display:

Breakpoints are indicated by the presence of red line drawn under the line in the editor at which the breakpoint is set. Here is how it occurs:

a) The *'setBreakPoint'* method is called to set the breakpoint.

b) The method finds the current caret position and gets the corresponding line element.

c) Once the line element is obtained a *'BreakpointAttribute*' is added to the *'AttributeSet'* element and the ChicEditor is asked to repaint itself. *'BreakpointAttribute'* is an object of type *'AttributeKey'*. A reference to this object is held in the ChicEditor.

The *'ClearBreakpoint'* method is called to clear a breakpoint. This method removes the *'BreakpointAttribute'* from the *'AttributeSet'* of the corresponding line element and asks the ChicEditor to repaint.

Undo/Redo [7]:

ChicEditor leverages the undo mechanism of Java 1.2 [22] to add the undo/redo facility. ChicEditor implements the UndoableEditListener [22] and holds a reference to a UndoManager [22] [8] [9]. When any part of the document is changed the *'undoableEditHappened'* method of ChicEditor is called. This method updates the UndoManager of the change. The execution of the undo and redo is carried out via calls to the *'executeUndo'* and *'executeRedo' methods* respectively. These methods call the *'undo()/redo()'* methods of the undo manager to effect the action.

Cut, Copy, Paste:

All these actions are part of the standard API of ChicEditor as a result of it extending the JEditorPane (Java API). The corresponding methods for the actions area *'cut(), copy and paste()'* . These methods are wrapped inside the corresponding *'executeXXX'* methods. For example *'executeCut()'* wraps *'cut()'*. The *'executeXXX'* are called to get the desired action. The constraint on this is that a text selection must be present for the action to take place.

Find:

This functionality is encapsulated inside the *'executeSearch'* method. The method gets the string to search for as it's argument. It then gets the caret position and searches the document after the caret position for the string. Once found the string is highlighted. Any subsequent calls to search will start from this point. The search is initiated by entering the text to search for in the JTextField in the ChicoryToolbar and hitting enter or right clicking on the JTextField and selecting a word from the list of words that we previously searched for.

3.5 The Popup menu:

ChicEditor holds reference to it's parent container i.e. the EditorPanel. The EditorPanel hold a static reference to an object of type EditorPopupMenu called popup_menu. The EditorPanel also implements PopupMenuListener interface. This is the interface through which the EditorPanel is notified of the selections in the popup_menu. Here is the sequence of events that happens when the user right clicks on the ChicEditor.

a) The EditorPanel of the ChicEditor is set as the listener to the popup_menu. The popup_menu is then made visible at the location of the mouse click.

b) When the user makes a selection, the EditorPanel is notified of the selection. The EditorPanel translates the notification to an appropriate *'executeXXX'* method call on the ChicEditor that it contains. For example when the user selects *'cut'* the *'executeCut'* method of the ChicEditor is called.

Chapter 4: The UML functionality structure.

4.1 Introduction:

UML [4][17] is an intuitive language that provides a developer tools to:

a) Pin down the requirements [26].

b) Design the class structure of a project/application.

c) Design the dynamic behavior of a class.

d) Design the interaction between objects.

Chicory is capable of letting the user do the above using UML. This chapter discusses the structure of the UML functionality and the interactions occurring inside. This chapter is not a *'User manual'* and does not provide operation instructions.

4.2 Basic structure:

The UML [4] tool (referred from hereon as 'tool') consists of four designers.

a) The UseCaseDesigner for Usecase diagrams.

b) The ClassDiagramDesigner for Class diagrams.

c) The InteractionDiagramDesigner for Interaction diagrams.

d) The StateDiagramDesigner for State diagrams..

Each of these designers extends a common class called as the UMLDesignManager. The UMLDesignManager in turn extends the GraphicDesign [25]. This means that the designers have the *'Rapid application development facility'* where a user can select beans representing UML constructs and drop them inside the designers to create live diagrams. Each of the designer has it's own beans which can be dropped ***only*** inside their appropriate designers to create the appropriate diagrams.

An important thing to note here is that the UMLDesignManager has it's own static TreePanel & root VisualNode variable. This was done have a separate tree and root node for the UML functionality. Doing otherwise would mean that all the UML functionality would appear in the Components tab under the Components tree.

It is important to note that the tool is driven from the UML tree and does not simply use the tree to represent the components. This means that we can cut copy paste and even customize components from the tree. The tree, due to it's tight integration with the tool thus provides a convenient way to serialize the tool.

4.3 Details:

What follows are class diagram and a detailed description of the structure. These are followed by significant interaction sequences in the tool.

The start of a new UML design case is signaled by the user

a) selecting *'new'* from the UML menu in the tools menu of ChicoryMenu.
b) Selecting *'new'* from the *'File'* menu or from the toolbar and then selecting the UML design case and clicking on ok.

Either of the above mentioned action causes the instantiation of a *'NewUMLCommand'* and it's execution by the ToolCoordinator.

The First batch of diagrams show and explain the hierarchical structure of constituent components of the tool. The second batch explain how the components work together.

(over….)

Batch 1: The Hierarchical structure:

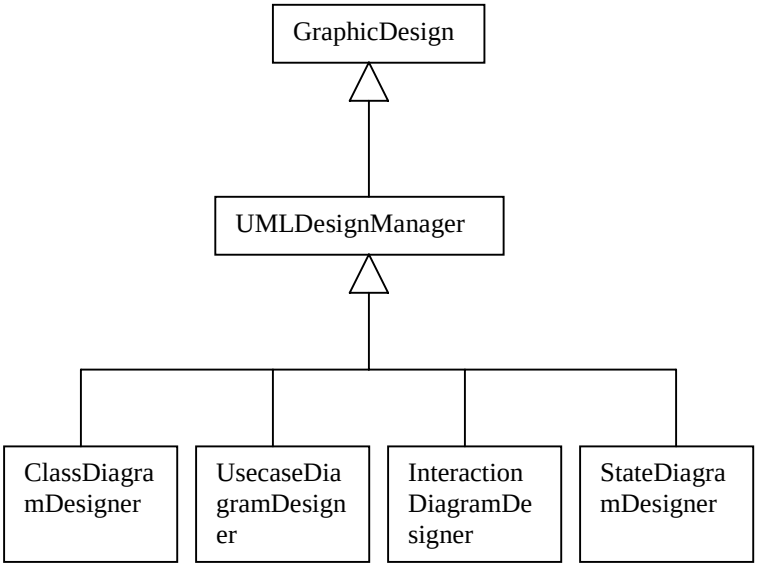


Figure:14

The Designer Hierarchy.

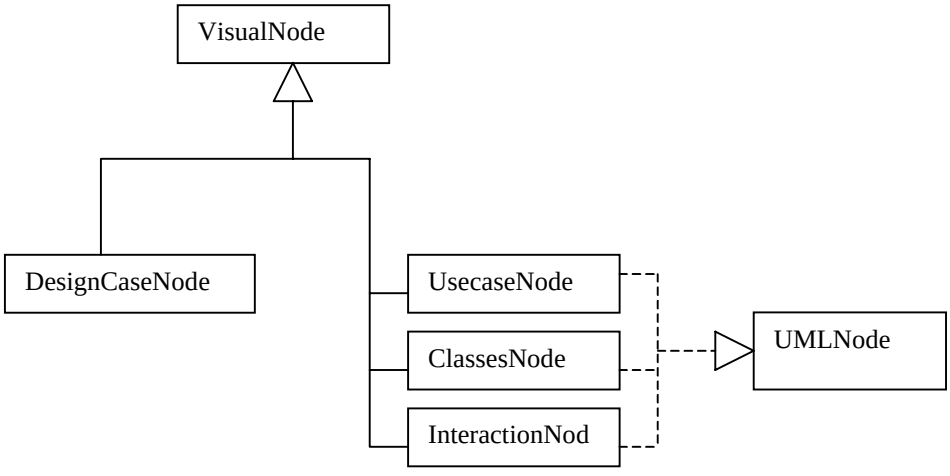


Figure :15.

The UML nodes.

4.4 DesignCaseNode:

The DesignCaseNode is the parent node of any UML design case. This node is referred to as the *'design root'*. This node will hold references of the three UMLNodes. The DesignCaseNode is created when a call to create a new Design case is made.

4.5 UMLNode:

A UML Node represents the starting point of each of the different diagram(save the state diagram) that we can make using this tool. Nodes that implement this interface hold a reference to the DesignCaseNode.

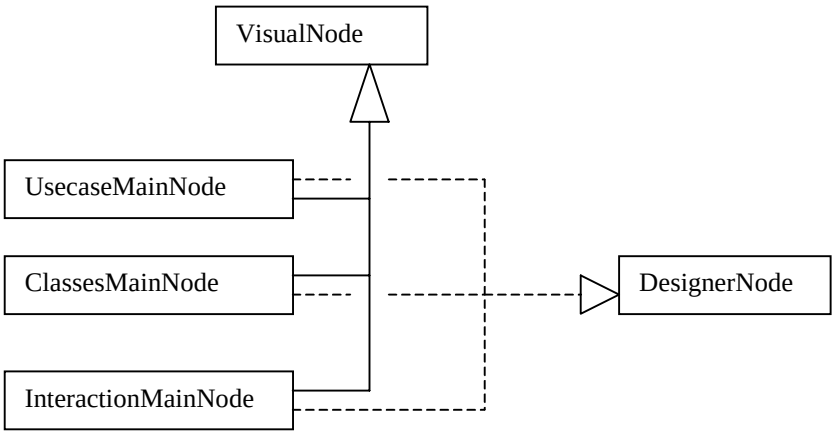


Figure 16.

The DesignerNode Hierarchy

4.6 DesignerNode:

The designer nodes are the nodes that contain the Designers. These nodes hold reference to the UMLContainer that holds the actual Designer.

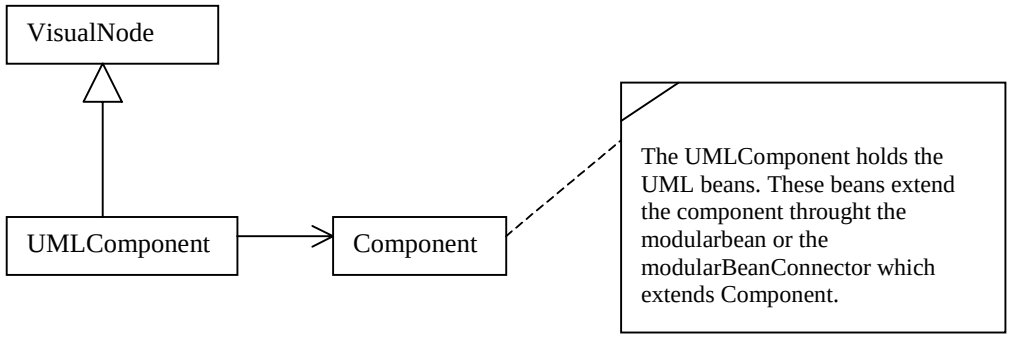


Figure: 17.

The UML Component node.

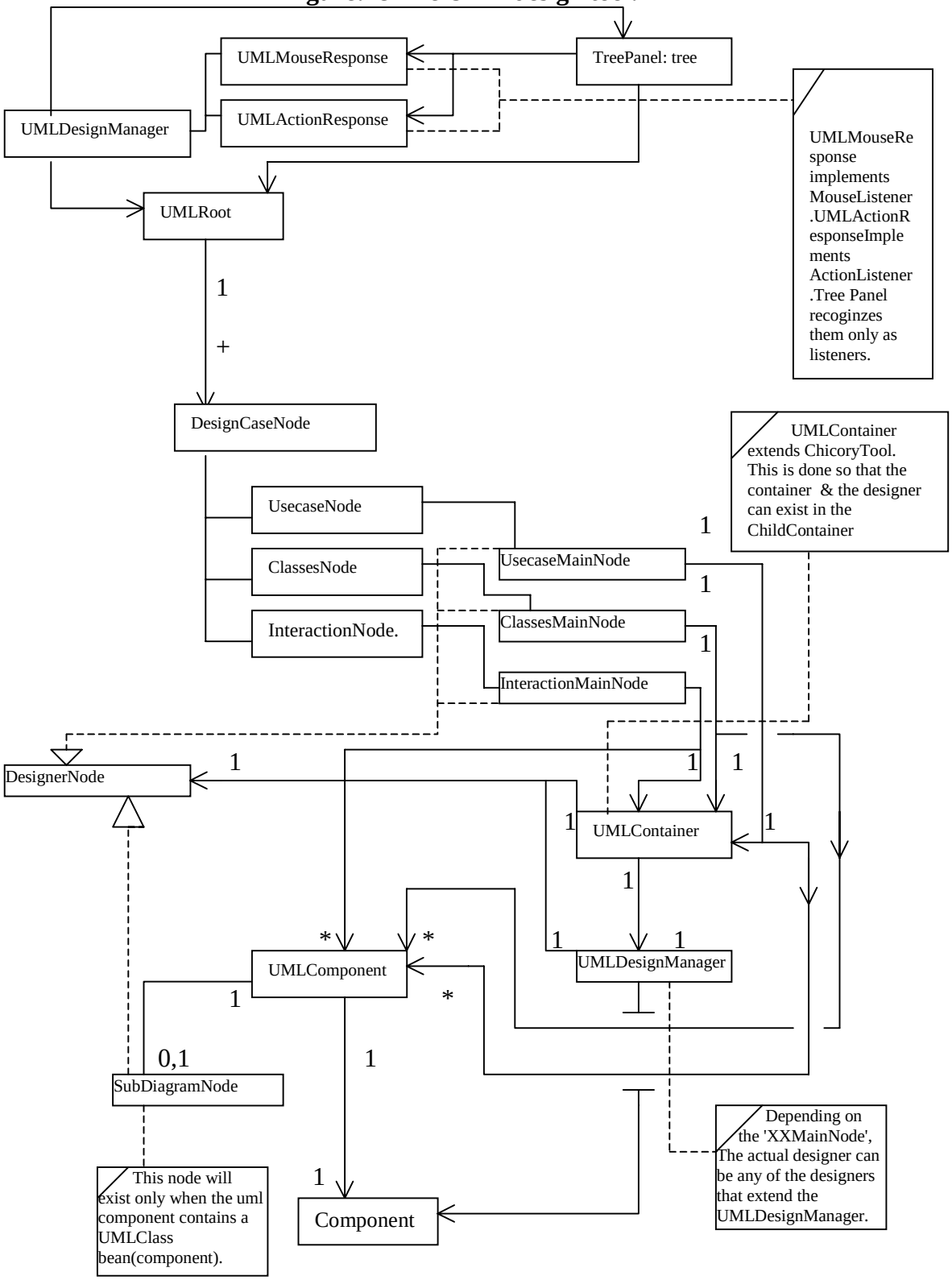
4.7 UMLComponent:

This node contains the UML beans for the tool. These node do not have child nodes except for the UMLClass bean which can have a SubDiagramNode as it's child. The SubDiagramNode contains the Designer for State diagrams.

4.8 Putting it all together:

This section explain how all the constituent components of the tool as explained above work together. The diagram that follows explains the functional structure of the tool. The diagram explain hierarchy through comments, where the need is felt.

Figure:18 The UML design tool.



4.9 How it works:

When the user signals for the creation for a new Design Case, the call is routed to the UMLDesignManager's *'constructDesignCaseNode'* method. This method is static. This method creates a DesignCaseNode ands returns it to the Command(i.e the user signal: NewUMLCommand). The command adds it to the tool's tree.

Here are the series of steps that occur when the design case node is being instantiated.

a) The UMLNodes are created and added as children to the DesignCaseNode .

b) When each of the UMLNode is being created, the appropriate XXXMainNode is created and added as child to the UMLNode.

c) When each of the XXXMainNodes are being created, the appropriate UMLContainer with the it's designer is instantiated and set as the element to the XXXMainNode.

A couple of points to note here are:

1) The Designers are intended to appear in the ChildContainer so that the user can select a bean and drop it in the Designer.

2) Since the designers are intended to appear inside the ChildContainer, they need to be in ChildFrames.

3) The UMLDesignManager contains a lot of code that is common to the Designers. Major amount of code handles the dragging and clicking on the components. Note that all the components inside the designer have rectangular bounds. Added to that is the fact that they are all transparent as they implement their own painting behavior. So If the components are overlapping the user can see a component below the topmost component. Now, if the topmost component is a connector then it's bounding box will occupy the area that the connector is drawn on. Thus if the user

wants to select a component that is visible through the bounding box, the connector will get selected instead. The figure below augments this explanation Note that 'D' is the connector. The figure explains the scenario if the extra *'handling'* code were not there. A,B,C,D extend component and hence will have rectangular bounds. The dashed rectangle shows the bounds of the connector D. Note that the connector is only a line but occupies the a rectangular area.. It is assumed that the connector is the topmost component.

Now if we tried to select 'C' by appearing to click on it, 'D' would get selected. The matters are further complicated when we consider that 'D' would still get selected when even if we clicked anywhere else on the bounding box for example, point X which has nothing below it and is not on the line of the connector.

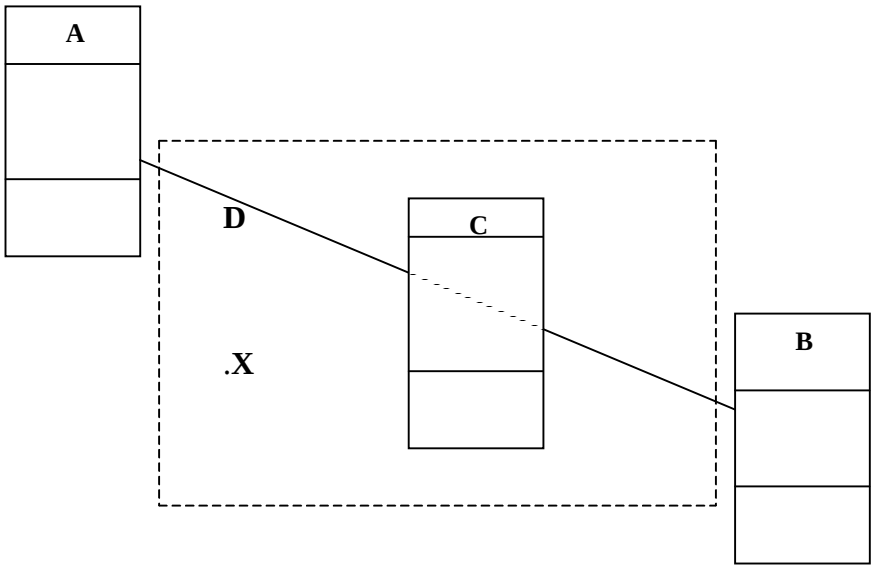


Figure 19: The Focus problem.

The problem is handled by the following strategy.

a) When we have a mouse click for focus, we gather all the component.

b) We go through the list of components and covert the click to the coordinates of that component.

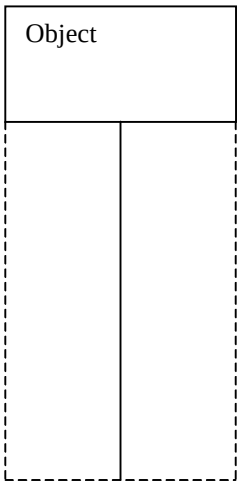
c) We then ask each component if it claims the click.

d) If it claims the click we give the component the focus. If no one claims the click the click was on a blank area.

e) The beans inside the designers are (functionally) either connectors or non connectors. While going through the components, if we come across a non connector, we check to see if the converted point is inside the bounds. If it is a connector, we know that it extends a modularBeanConnector. We call the *'containsPoint'* in the modularBeanConnector class.

f) Special considerations exist for the Interaction diagram Beans. One of the beans is the IRObject bean, which is the representative of an object inside the InteractionDiagram. The bean looks like the following diagram. Note that here the representation is not rectangular(the dashed line is the bounding box) and the component is not a connector. Special code exists inside the InteractionDiagramDesigner to deal with the situation.

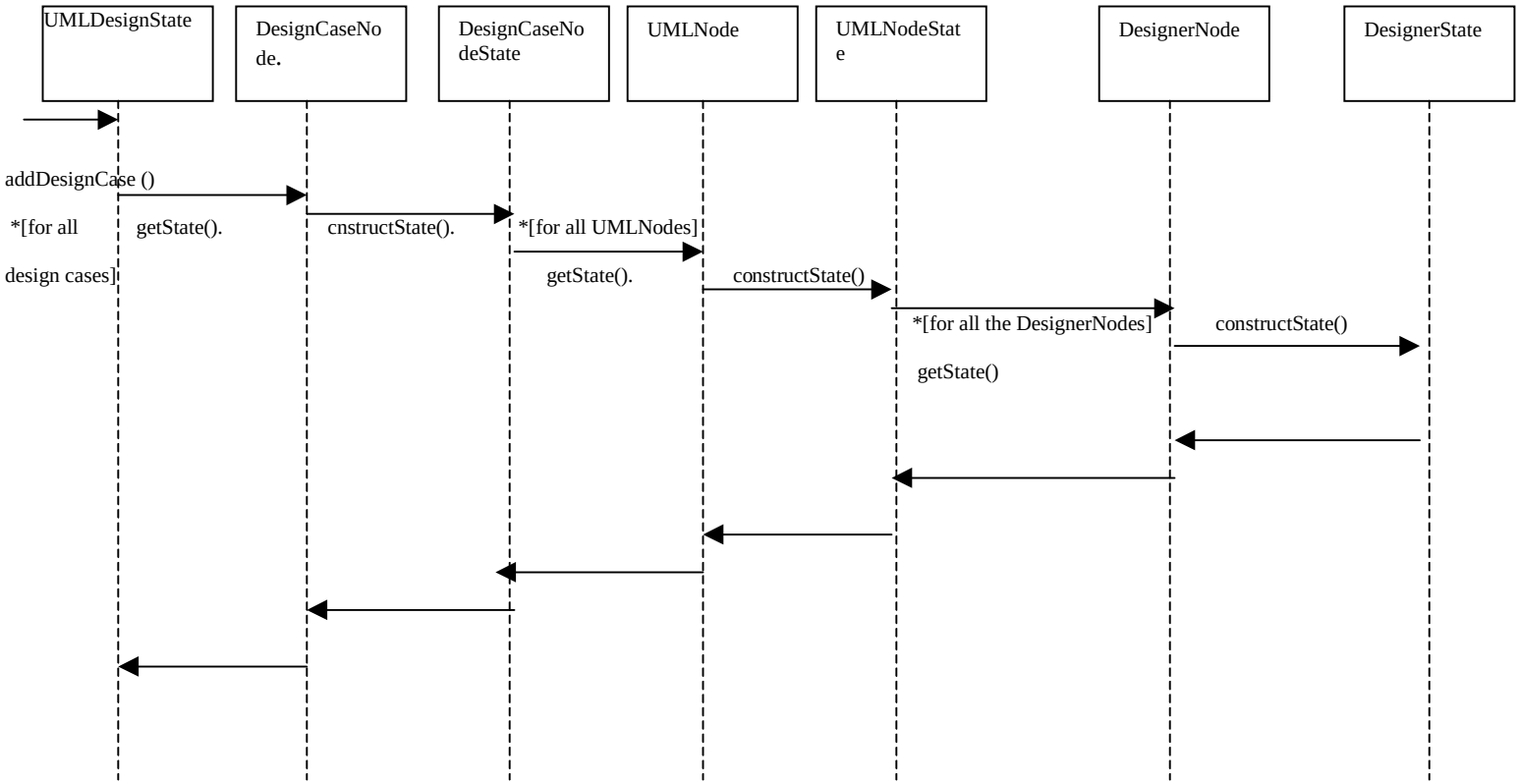
Figure 20:The Interaction Object.



4.10 Serialization: The following diagram shows the interaction occurring during the serialization of the UML design tree.

The serialization process is started by creation of a UMLDesignState object and calling the *'addDesignCase'* method on the object with each design case in the tree. Please refer the diagram following this interaction for the hierarchy of the state objects.

Figure 21: The UML serialization process



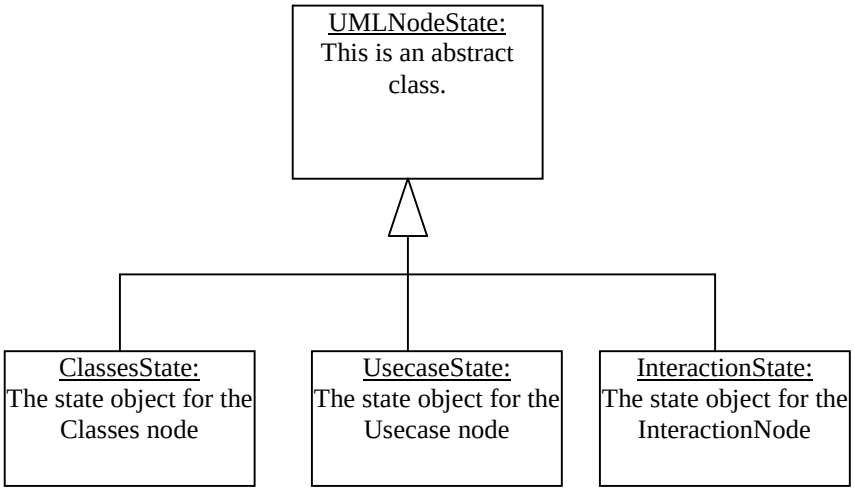


Figure 22: Hierarchy for the UMLNode state objects:

Each uml node returns the appropriate state object.

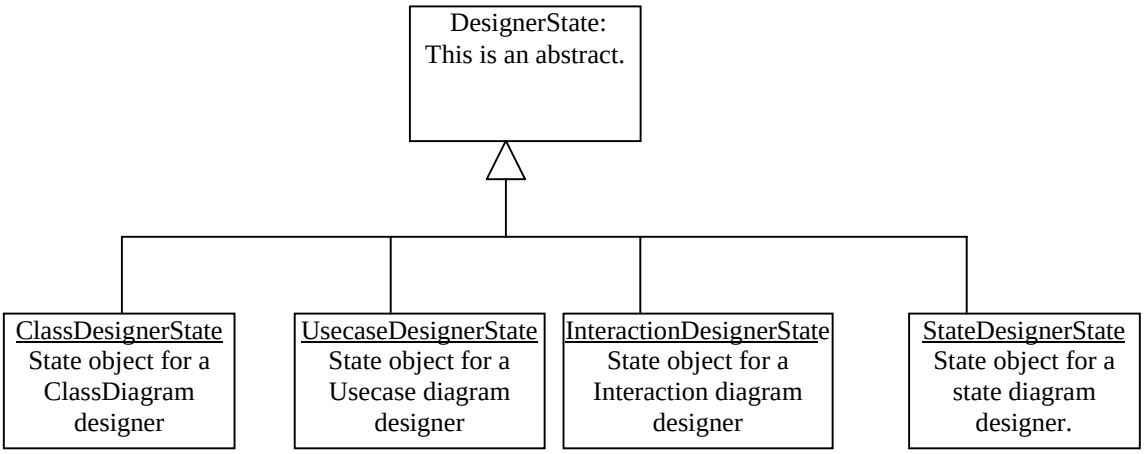


Figure 23: Hierarchy for the designer state objects:

Each DesignerNode will return the appropriate State object.

4.11 Deserialization:

The Deserialization process is carried out in the exact opposite sense of the serialization state. Here is how it happens.

a) When deserializing, we obtain a UMLDesignState object. We pass the current UML design root to it using a call on the *'constructDesignTree'*.

b) In the method we go through the list of deserialized DesignCasNodeState objects and instantiate a DesignCaseNode with the DesignCaseNodeState and the design root as the constructor parameter.

c) When the DesignCaseNode is being instantiated, it obtains the UMLNodeState objects from the DesignCaseNodeState objects and creates the UMLNodes accordingly using the UMLNodeState objects & the DesignCasNode as constuctor parameters.

d) As each UMLNode is being created, it goes through the list of DesignerState objects in the UMLNodeState object and creates a DesignerNode with the DesignerState object & the UMLNode as the constructor parameters.

e) Note that each time a node is create in any of the above steps, it is added to the UML design tree. For example when the DesignCaseNode is created it is added to the root.

f) When the UMLNodes are created each of them is added to the DesignCase node and so on. This recreates the whole UML tree.

Chapter 5 :The UML Beans

5.1 Introduction [4]:

This chapter explains the structure of the UML beans and recommends guidelines to be followed when making additional UML beans. Before continuing with the explanation we must emphasize the following facts:

a) All the Uml beans extend the same super class (i.e. modularBean).

b) All the Uml beans are lightweight and implement their own drawing behavior. Hence the part of the bean's bounds that is not drawn upon, appears transparent.

c) modularBean extends java.awt.Component hence the Uml beans will have a rectangular bounds. This means that the beans will be rectangular without appearing to being so.

d) All the Uml beans are Java beans [23] [2] too and hence can operate in that component model

5.2 Basic Structure:

The Uml beans fall strictly into two categories. These are

a) Connector: In which case the bean extends the modularBeanConnector.class (the modularBeanConnector in turn extends modularBean). For example: umlAggregation, IRConnector, and Transition.

b) Non Connector: In which case the bean extends modularBean.

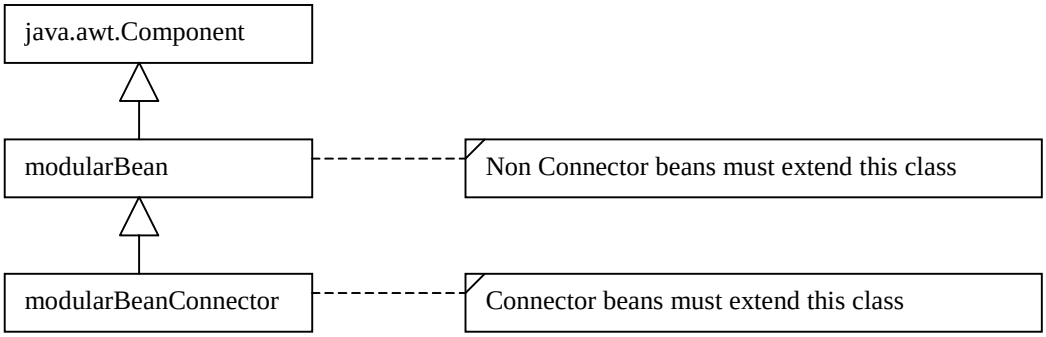


Figure 24: UML Beans Basic Structure.

Note that the modularBean as well as the modularBeanConnector are abstract classes. This means that for each of the bean that extend either of the above, some methods require implementation. Here is a list of the methods that need implementation.

a) modularBean

1) *measure*: This method should be used by the bean to measure itself and set the minimum size.
2) *getMinumumSize*: should be used to return the calculated minimum size. Note that the getPreferredSize in the modularBean is coded to call *getMinimumSize*.
3) *Paint*: should be used to render the component on the graphics context.

c) modularBeanConnector:

1) *setSource*: should be used to set the source of the connector i.e. the starting bean of this connector.

2) *setTarget*: should be used to set the target of the connector i.e. the ending bean of this connector.
3) *CheckRules*: should be used to check if the source and target are valid for this connector, for example an IRConnector is limited to having IRObject as the source and the target.
4) *removeSelf:* the behavior to be implemented when the connector is removed.

Here are a couple of more points that need to be considered when making connector beans.

a) As was explained in the chapter that explained the UML designer, the bean bounds are rectangular even though their appearance is not. This leads to the problems of
   1) Determining the target of a mouse click (discussed previously). The *containsPoint* method returns true if a point is contained inside the connector. The point is considered to be in the connector if the point is on the line. The developer is advised to view the source code of the previous connectors to understand the implementation of the method and verify if the implementation applies to the bean being developed.
   2) Focus indication: The component having focus is indicated by drawing squares on the corners of the bounds of the bean. This however creates problems for a connector. The connector is a line and thus has only two end points as against four. This means that for a connector only the two end points should have the squares indicating the focus. The code to draw the squares is present in the UML designer. The code however needs to know where to draw these squares. The

getConnectorLine method should return the line that is the connector. The focus indicating code then draws the squares at the end points of this line. This problem is illustrated below.

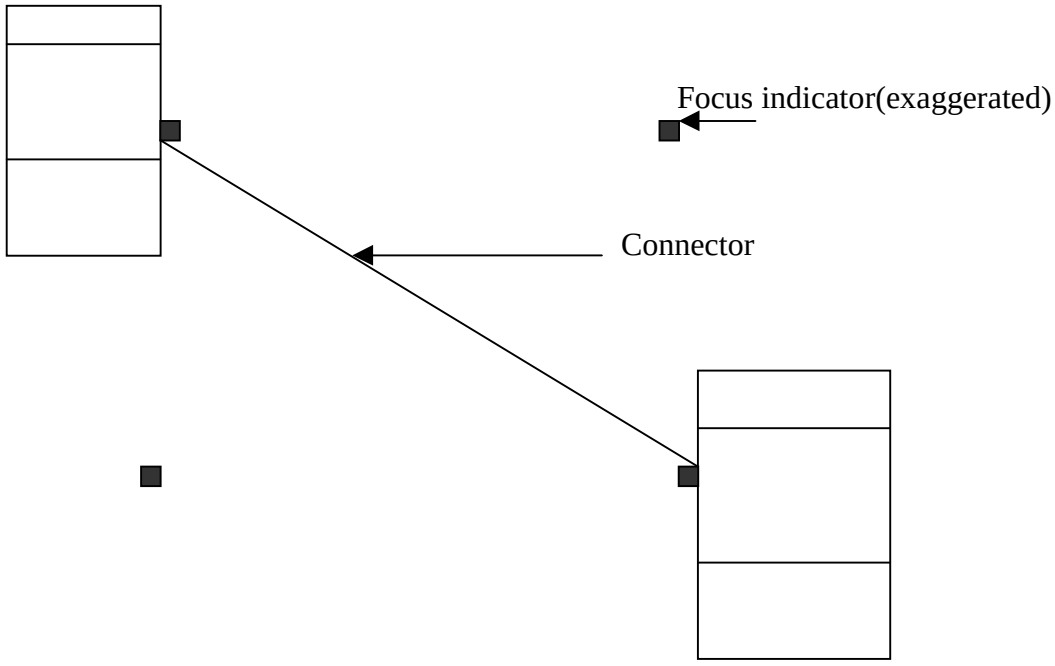


Figure25:Wrong indication of focus for a connector.

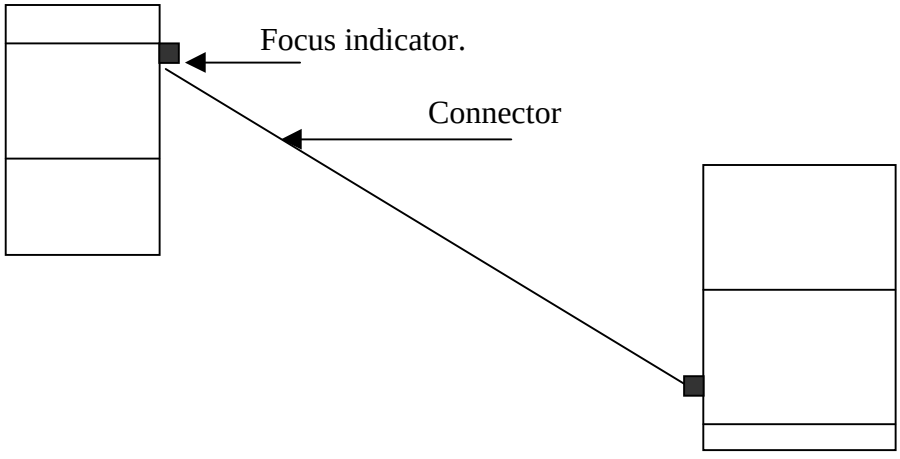


Figure 26: Right indication of focus for a connector.

5.3 Suggestions for building UML beans:

a) Create the API specification files for modularBean.java and modularBeanConnector.java using Javadoc and be familiar with the method calls.

b) Be familiar with the source code for the two files. Understand the sequence of operations. Look at any one of the concrete implementations. umlAssociation.java is a good choice to understand the logic.

c) It will be advisable to have concrete ideas of how the bean should look.

d) Ensure that the bean fits into the scheme for working the problems of target selection and focus indication.

e) Be aware that non-connector beans that have non-rectangular representation must override the *contains* method of the java.awt.Component to solve the above problems.

f) Implement the valid connection rules for a Connector in umlRules.java.

5.4 List of Currently existing UML beans:

Class Diagram beans.

umlClass, umlAssociation, umlAggregation, umlComposition, umlConnect, umlDepend, umlGeneralize, umlNAssoc, umlNote, umlPackage, umlRealize.

Interaction Diagram:

IRConnector, IRObject.

State Diagram:

State, Composite, Transition, Start, End.

Usecase diagram:

Actor, Usecase, UsecaseConnector, UsecaseGeneralize.

Chapter 6: The Debugger.

6.1 Remote Debugging [2]:

The Debugger API is built around the concept of *remote debugging*. This concept implies that not only is the debugger running in a separate process than the debuggee, but it also may be running on a separate machine. This setup offers great flexibility. Besides the obvious benefits of being able to debug from a distance. The Java application may be running on a resource-challenged machine such as a PDA, a Set-top-device. This remote machine may have small amounts of memory, slow CPUs, or small screens, among other things-definitely not a worthy machine for a developer to use for debugging purposes. With remote debugging, the developer can stay within his or her normal development environment while debugging a Java application. Remote debugging breaks the debugger into several parts. There is the debugger client, the debugger server, and a communication protocol. The debugger server resides in the target-usually code inserted into the target process or perhaps embedded in system software. The debugger server performs the important low-level work of the debugger. The basic functionality of the debugger server is to control the debugger and obtain information on its internal state. The debugger client is the part of the debugger that the developer will interact with.

6.2 Debugger basics [18] [19] [20]:

Remote debugging requires two main parts.

a) The Debugger server : This is the part that will actually run the application that we want to debug . It will also perform a series of operations on the application on the application based on the requests that a client sends it.

Java achieves this in the following way:

The java virtual machine(jvm) that runs any java program can be started in two modes i.e. the normal mode and the debugging mode. In the debugging mode the jvm [16] is started with *'-debug'* option. When the Java Virtual Machine [16] is started in debug mode-by supplying the `-debug` switch an extra thread is spawned; it runs a nonpublic class called `sun.tools.debug.Agent`. This class implements the `Runnable` interface and runs in a thread named "Debugger Agent." The `Agent` class handles the communication with the debugger client through the socket and also performs much of the execution of the debugger's commands. It also obtains inside information from the Java Virtual Machine via a set of native methods that are implemented in the shared library named *agent* (`libagent.so` on Solaris; `agent.dll` on Win32).
The `Agent` class also makes use of several of the other nonpublic debugging classes , most notably the `BreakpointHandler` class. The `BreakpointHandler` class also executes within another thread named Breakpoint Handler. This thread is contacted when actual breakpoints occur; thus being in its own thread allows it to contact the Agent thread in an asynchronous manner. The `Agent` class can then pass the information back to the debugger client. A third, less-important thread also exists. The `EmptyApp` class contains a single static `main` method (a simple Java program), which is executed as a placeholder until the real target application is started. It simply lives in a suspended state.

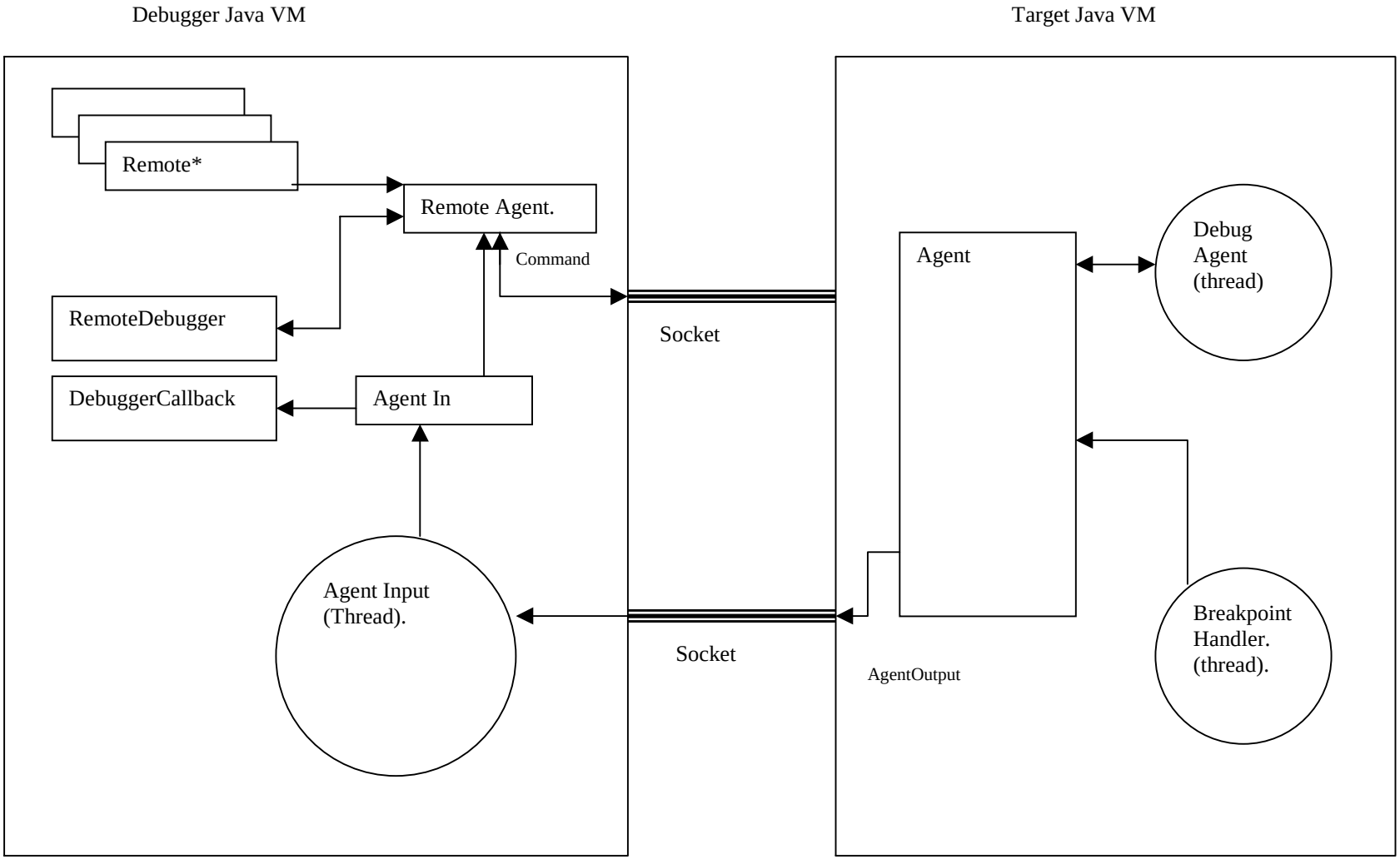
b) The Debugger client :The debugger client is the program with which the user interacts; it drives the target Java Virtual Machine(Debugger server). The presence of the Java Debugging API makes the task of controlling and getting information from the target Java Virtual Machine possible. The Java Debugger API consists of a

handful of classes and a single interface.The Debugger API performs a number of tasks on behalf of the debugger client. It manages the communication to and from the debugger server. This communication occurs over two socket connections made to the debugger server. One socket is used for sending client requests to the server. The other socket is used for receiving notification events from the server. The requests are synchronous actions initiated by the client-such as the client asking the server for information about the debuggee, or asking the server to perform tasks such as setting a breakpoint. The notification events are asynchronous to the client. That is, the client does not know when they will come, and the notification events may actually arrive while the client is performing requests. The debugger client invokes the methods defined in the public classes of the Debugger API . The Debugger API then translates these method calls into command messages and sends them to the server over one of the sockets. The debugger client simply blocks on a method call while this occurs. The debugger server then fulfills the request and simply acknowledges it, or sends information back to the client in a reply message over the same socket. The Debugger API converts the reply into an appropriate return value for the debugger client. All of this communication is handled by the `RemoteAgent` class. The `RemoteAgent` class is non-public and is never directly accessed by the debugger client. Notification events are implemented with a callback mechanism. The debugger client implements the `DebuggerCallback` interface (described in the next section) and registers the callback with the Debugger API. Once this registration is complete, the debugger client does not need to perform any other actions. The methods defined by this interface are invoked when a notification event occurs. The Debugger API-during

intialization-creates a thread. This thread is named "Agent Input" and its only task is to read messages from one of the sockets-the notification event socket. When a message arrives from the debugger server, that message is interpreted and the appropriate method of `DebuggerCallback` is invoked. .The communication between the debugger client and server over the two sockets occurs via a simple message protocol. The messages are composed of a simple command ID followed by optional data specific to the command. These command IDs are defined in the `AgentConstants` interface. This interface is not public, but you'll notice some of the public classes do imple-

ment it. The Debugger API is initialized by the debugger client when the client instantiates the `RemoteDebugger` class. When this class is intantiated the client passes the callback object to the Debugger API. At this time the two sockets and the "Agent Input" thread are created. The call-back object is any object created by the debugger client which implements the `DebuggerCallback` interface.

Figure 27:Debugger Architecture



6.3 Chicory Debugger:

Chicory leverages the Java debug API to debug an application. The basic scheme of the Chicory debugger is as follows.

a) The ProjectManager holds a reference to a RemoteDebugger object to initiate the Debug API.

b) When Chicory is started and a new Project is created. The RemoteDebugger is spawned with the with the project directory as it's Classpath and the verbose option set to false. The ProjectManager implements the DebuggerCallback[18] interface and hence is passed as the *'DebuggerCallback'* to the RemoteDebugger. This creates a jvm in the debug mode. The RemoteDebugger will wait till it is asked to run a particular class.

c) When we want to debug a class, we enter the enter the debugging mode by selecting the *'Start Debug'* either from the Menu or from the toolbar. If we are debugging for the first time after startup or we changed the ChildFrame having focus, we show a dialog asking the user for the class to debug. When we get the class name. We call the *'run'* method on the RemoteDebugger with the class name as one of the parameters.

d) The execution of the program can be controlled by setting/resetting of break points(discussed later). When the program hits a breakpoint the execution of the program is halted at that point. Also since the ProjectManager implements the DebuggerCallback[18] interface, it is notified of the breakpoint via the method *'breakpointEvent'* method with the current RemoteThread [18] as the parameter. We use this as the starting point to gain information about the current status of the program in it's halted state. Using various *'getXXX'* methods we obtain the global

variables, local variables, current thread group, the stack image and the breakpoints . This information is processed and sent to the Graphical Display(discussed later) after properly formatting it. The methods used to obtain information mentioned above are *'getLocalVariables()', getGlobalVariables', getStackImage()', 'getCurrentThreadGroup()' and 'getBreakpoints()'* of the ProjManager.class.

e) After a program hits a breakpoints, we can control the program execution using the the following options

1) Step into : Makes the program step into a method call and halt. If the current line is instruction, the instruction will be executed and the program halts again.
2) Step over : Make the program execute the method call and halt at the next line.
3) Continue : Continue running normally.

At steps 1,2 & 3the Status (see 'd') is updated .

The above mentioned commands can be invoked from ChicoryMenu or from ChicoryToolbar. These commands are routed to the ProjectManager's(ProjManager.class) *'actionPerformed'* method. From here appropriate calls are made to the RemoteDebugger using the Java debug API.

6.4 Setting/Resetting of Breakpoints:

The setting or resetting of breakpoint is carried out from the ChicEditor. Hence ChicEditor is tightly integrated with the process of breakpoint setting. Before we show the working of the breakpoint setting/resetting, we need to look at an Object called as the FileManager.

FileManager is used to keep a track of the files that the 'ChicEditors' send it. The file (actually the 'ChicEditors' sends a reference to themselves) is sent by a 'ChicEditor' when the user violates the file that it has opened. The file is violated when the user performs a save or changes the status(sets/clears) of the breakpoints. This is necessary as the Debugger needs to be updated of this change to keep track of the breakpoints.

The following interaction diagram shows the processes of Setting a break point. Clearing a breakpoint and the process of updating the debugger. The process of saving a file and compilation are covered in the Chapter that covers the text editor.

The setting/clearing of breakpoints is initiated from

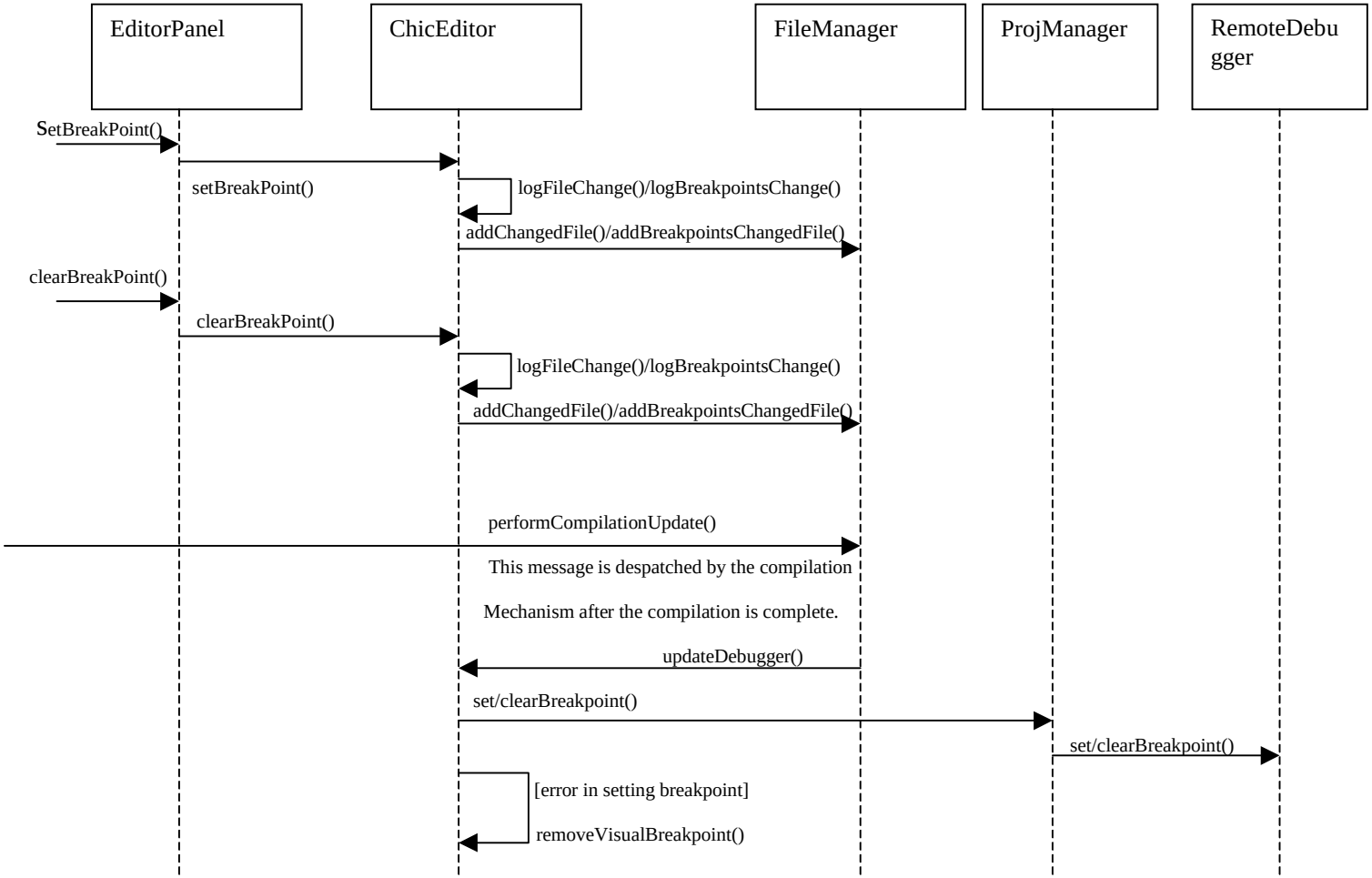
a) ChicoryMenu

b) ChicoryToolbar

c) The popup menu that the user will get on right clicking on a ChicEditor.

All of the above action result in calling the *'setBreakpoint'* or *'clearBreakpoint'* methods of the EditorPanel of the currently selected ChildFrame.

Figure 28: Setting/Resetting breakpoints.



6.5 The Debugger user interface:

Chicory provides a graphical user interface to interact with the debugger. Shown below is the class diagram for the user interface. The user interface is in a file called DebugArea.class. Note that the DebugArea hold references to one or more DebuggerDisplayHost type variables. The DebuggerDisplayHost in turn holds reference to a single instance of the Display interface. Note that in the future we may decide to add to the information being displayed in the DebugArea. In this case all that needs to be done is to make the class that is going to display the information, implement the Display interface.

Now all that needs to be done is to add an additional DebuggerDisplayHost variable inside the DebugArea. The values can be routed to the Actual display (the class that implements the Display interface) using the *'showValues'* method in the Display interface.

Figure 29: The Debugger UI

Chapter 7: Optimization

7.1 Introduction [5] [6]:

Java has a reputation for being slow compared to native code (as produced by languages such as C/C++). However a good understanding of the Java language can lead us to significant improvements in speed. The process of trying to improve the performance of an application is called as Optimization. This chapter discusses some of the techniques used to improve the performance of Chicory. As we proceed through the chapter we will notice that there is always a time (execution speed) v/s space (memory usage) tradeoff.

7.2 Basic concepts:

The following table shows the time taken in nanoseconds to perform operation for a standard Java VM. The source of this table is Java Report May 1998 volume 3, Number 5. The article is *'The need for speed, Optimizing your Java Programs'* by Alex McManus and John hunt [6].

Operation	Java	Java + JIT	Native Code
Local Variable assignment	67	10	5
Instance variable assignment.	280	14	5
Method call.	541	40	40
Synchronous method calls.	1767	1215	2053
Object creation	2189	2250	1701
Array creation	178	14	13

We can make the following observation from the table.

1) Local variable can be accessed faster than a variable of any other scope.

2) Methods calls are relatively costly. Synchronous method calls slow down execution tremendously.
3) Object creation is tremendously costly and could increase more with addition of more code inside the constructor.
4) In cases of non-JIT use, array creation timings hint at reuse rather that instantiation.

It is with these observations that we started out to optimize Chicory.

The following sections illustrate the techniques that we employed to increase the performance of Chicory.

7.3 Optimization techniques [6]:

1) Use of a JIT compiler: Use of a JIT compiler increases the execution speed of a Java application significantly. We recognized this aspect and used the JAVA_COMPILER system variable (Windows NT) set to *'symcjit'*. This caused the JIT to be used increasing the performance. This approach had to be adopted with the earlier beta releases (2 and 3). By the 4th release the JIT was implemented as default in JDK.
2) Use of the Java extension Mechanism: The reader is advised to refer to the chapter on the Extension mechanism available at http://java.sun.com/docs/books/tutorial/ext/index.html for detailed information. Here is how we used the extension mechanism for improving performance. The current release of Java allows application to be packaged and run directly from jar files. When an application is run from the jar file, the Class that starts that starts the application looks for the classes that it needs from the *'Class-Path'* attribute of the manifest file of the jar. We used a startup jar called as *'Chicory.jar'*. This jar had the location of the jar containing all the classes needed to run chicory in it's *'Class-Path'* attribute. The advantage that we gained was that the

search for classes was limited to the classpath specified in the attribute so the classes could found faster thus making the execution faster.

3) Use of ’Optimize' setting on the Compiler: The compiler that Sun ships with Java can be invoked by using javac followed by options followed by the source files. One of the options that can be used is the '-O' option. When the compiler is invoked with '-O' option, it inlines methods which are private and final. This means that code will run faster but the class files will be larger. We used this approach to convert methods used by a class internally to be private and final. We then compiled the source file using the '-O' option. This translated into improved performance where we had repetitive calls to internally used methods. By default the *'javac'* the java compiler creates class file containing line number information. If the code was compiled using the *'-g'* option, it would also contain variable debugging information. This extra data swells the files. The usage of the optimize flag removes this debugging information. It may also remove some redundant code. This causes compact class files.

4) Method call elimination: From the table we understand that access to local variables and local variable assignment is much faster that a method invocation. We searched for methods that were accessor methods that were called repeatedly. We replaced the calls by making the method calls once and assigning the returned value to a local variable. We applied the same remedy to methods that performed intensive calculations using variables that were obtained using method calls.

5) String buffer use: When a program deals with manipulating strings i.e. say adding strings to create the required string, the best API to use is the StringBuffer API. The StringBuffer provides a method called *'append'* that adds to the existing buffer. The

buffer can be converted to a string by calling it's *'toString'* method. This better than adding one string to a second string to create the string where a new string is instantiated (costly operation) at each new addition. We recognized this advantage an applied the StringBuffer in appropriate situations to improve performance.

6) Avoiding Object instantiation: From the table we know that object instantiation is a costly operation. A couple of places where this can affect the performance is creation of new vectors or storage structures and firing of custom events. We tackled the situation by

   a) Reusing vectors wherever possible. Note that it is not advisable to reuse a vector containing too many elements as removing all the elements can be slower than instantiating a new one.

   b) Holding a reference to an event that requires to be fired. This results in using the same event time and time again thus saving instantiation. Note that we see if the event is used simply as an indicator or a carrier of information. In case that the event is used only as an indicator, the above strategy can be easily implemented. In case that the event conveys information, we need to provide methods to set the appropriate information in the event. This leads to faster performance event though additional data setting method calls are implemented.

7) Avoiding synchronized calls and re-writing library classes: Methods that are Synchronized are tagged by adding a *'synchronized'* keyword in their signature. We observe from the table that calls of this are more time consuming than the regular method calls. This time delay can prove to be a bottleneck when we consider a functionality where

a) A dynamic data structure is needed.

b) Operations are performed on the data structure repeatedly.

An obvious choice for this kind of operation could be the java.util.Vector.class. The problem here is that most of the commonly used methods of Vector are synchronized. This means that, thought the Vector might perform satisfactorily for small amounts of data, there is severe performance degradation when large amounts of data are involved. We worked around this class by building a dynamic data structure called ObjectBuffer that performs operation similar to the Vector but does not contain synchronized method. We do point out that the disadvantage to doing this the fact the data structure is not thread safe.

8) Vector Instantiation: Vectors can be instantiated without an initial size. Since the Vector is dynamic data structure, Objects can be added to it at run time without worrying about running out of bounds. Thought this is an advantage, the problem lies in the fact that the Vector doubles in size every time it runs out of allocated space. This means that the time spent in allocating space increases as we do a rapid addition of a large number of Objects to the Vector. Recognizing this problem we modified code to initialize vectors with initial sizes and specified the increment to be carried out on running out of space.

Chapter 8: Future Work.

This chapter explains the functionality that is planned on being added to Chicory.

While planning ahead we have taken care to emphasize Chicory as the base platform into which all the tools will be integrated. This chapter explains conceptually the tools that are going to be added and the effect that they will have on Chicory as a whole.

8.1 Design Patterns:

Design Patterns in computer science are a literary form of problem-solving software engineering which attempts to guide our application development through an understanding of how we build and interact with computer programs. The goal of patterns within the software community is to create a body of literature to help software developers resolve common, difficult problems. Patterns also provide a common vocabulary for communicating insight and experience about these problems and their solutions. With this shared vocabulary and documentation it is easier to convey the architectures and mechanisms behind our object-oriented designs.

To this end, patterns are summarized in documentation templates to capture these experiences. Patterns also have been derived from programming idioms and the documentation of best practices and lessons learned.

8.2 Why use Design patterns:

Designing object-oriented software from scratch is difficult; reusable software is even more difficult. Issues such as the granularity of classes, interfaces, inheritance hierarchies, relationships, etc. confound the novice designer and present recurring questions to the experienced developer. Secondly, expert designers do not know how to solve every problem from a blank sheet of paper. They reuse solutions, idioms, and techniques that have worked in the past (hence their experience). Patterns excel in this respect because they solve particular, recurring design problems. A designer familiar with a collection of patterns can apply them immediately with less time spent (re) discovering them. Overall, object-oriented designs become more flexible, elegant, and finally more reusable when developed with patterns in mind.

8.3 Design patterns and Chicory™ :

Chicory already has a robust architecture for handling tools based on the drag and drop paradigm. This means that Chicory is well equipped to deal with Java Bean™ components.

This sound foundation allows us to plan for a tool that allows a developer to rapidly build architecture for a solution, using design patterns.

Here is what we plan:

a) Well known patterns can be encapsulated as Beans. These beans can be held in a tool palette.
b) We can design a RAD like to tool extending the Graphic Design class to form the host for the beans. After putting the beans in the designer, the developer can then connect the beans, thus connecting the patterns. This will form the structure of the application that the developer wants to design.
c) After we have the structure ready, we can generate code from the structure. This code will of course be skeletal in nature with the user's input necessary to provide the implementation for the methods.

By designing and integrating this tool into Chicory, we plan to move Chicory™ to being and extremely efficient software development environment.

8.4 The Refactory tool:

Refactoring is a term used to describe techniques that reduce the short-term pain of redesigning. When we refactor we do not change the functionality of the program, rather we change its internal structure in order to make it easier to work with.

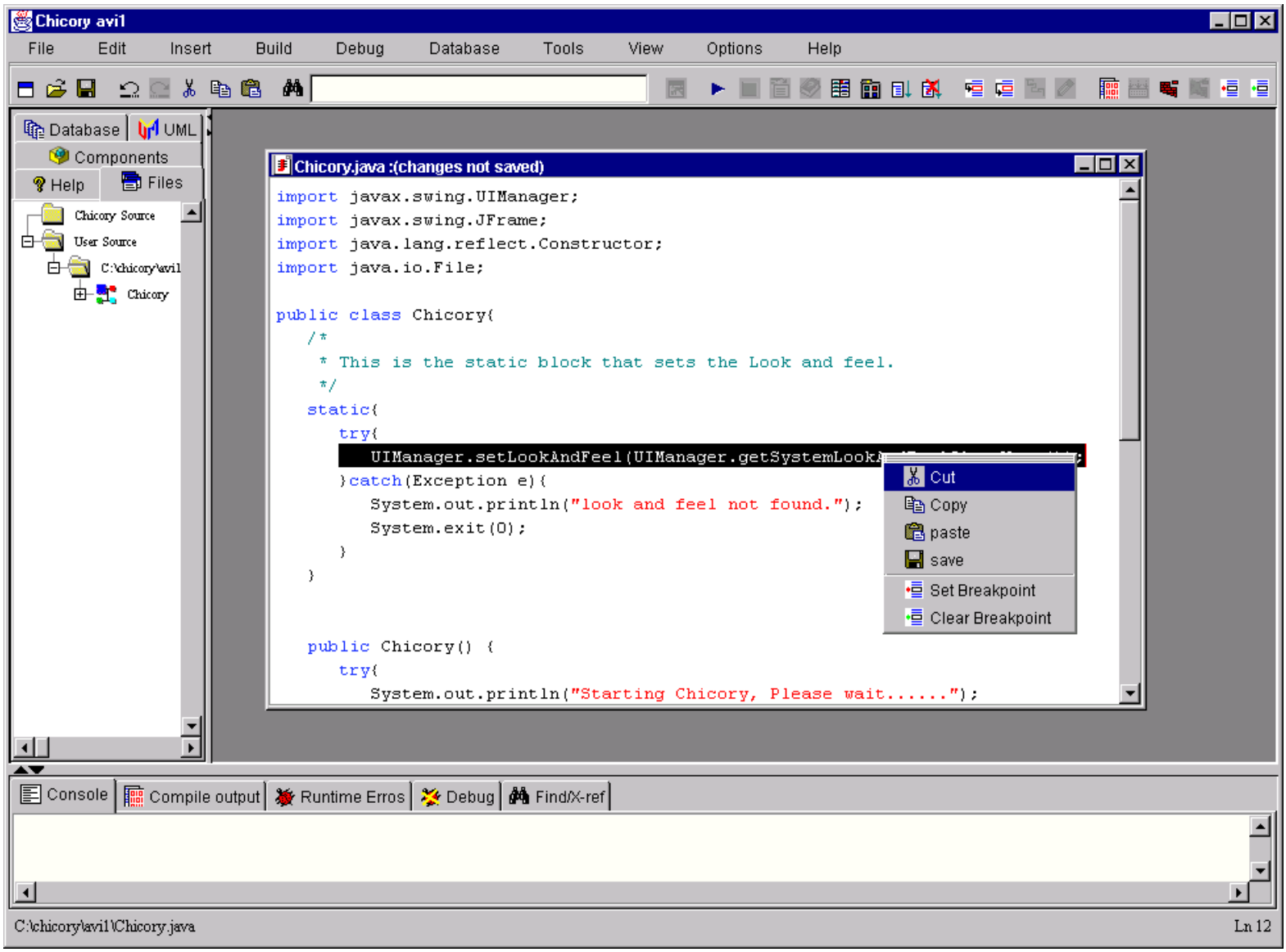
Refactoring changes are usually small steps: renaming a method, moving a field from one class to another, consolidating two similar methods into a super class. Each step is small, yet a couple of steps can do a world of good to a program.

Since refactoring occurs at all levels within the software development life cycle, the ability to perform refactorings automatically is crucial to software evolution. This is especially true with the advent of design patterns. Due to the relatively recent development of design patterns, few existing programs use the flexible designs typified

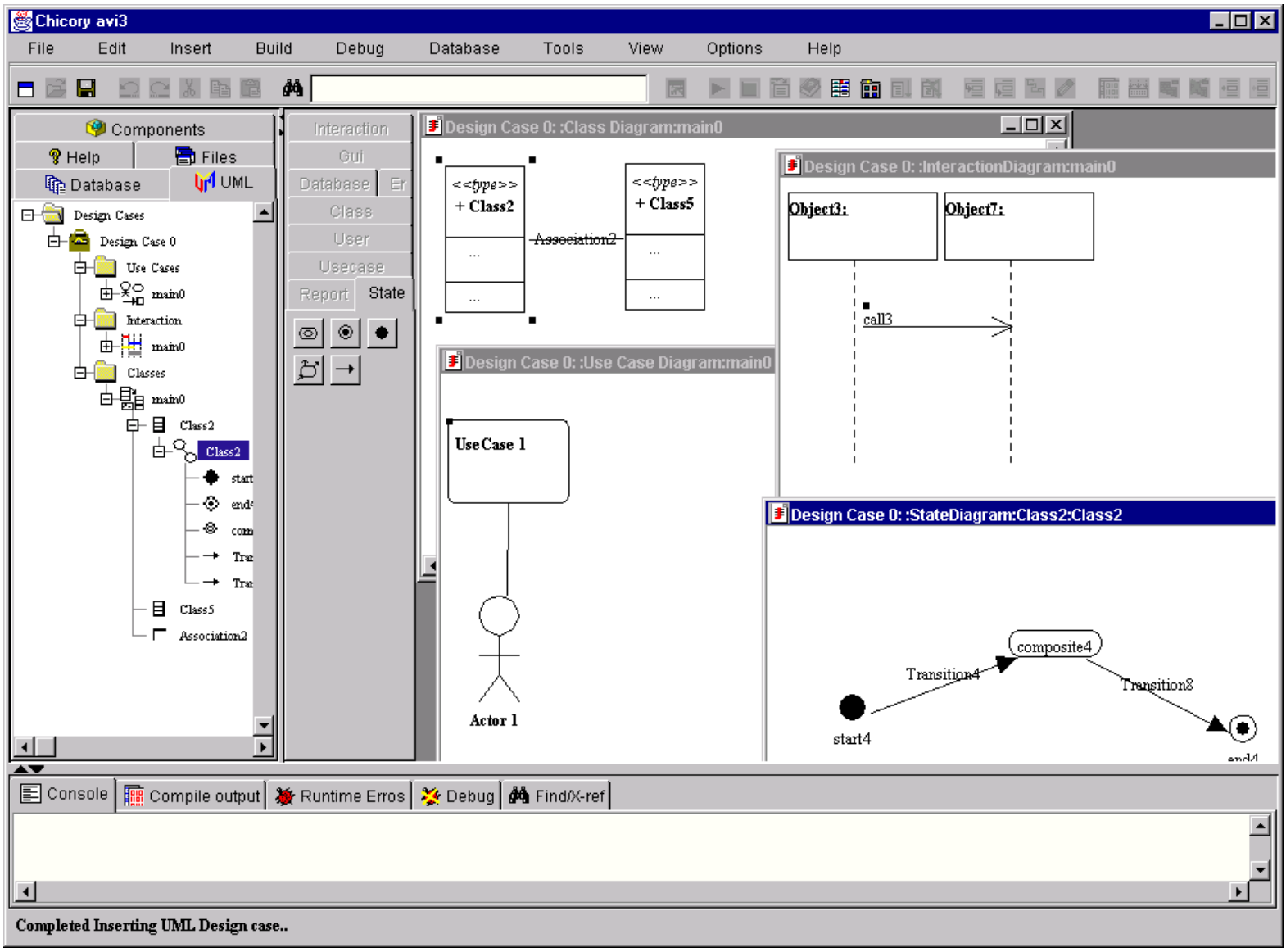
by them. Adding these designs to existing software can be a tedious process. Refactorings simplify this process by automatically handling the details of the code.

Thus the design and integration of this toll will add tremendously to the software engineering muscle of Chicory™.

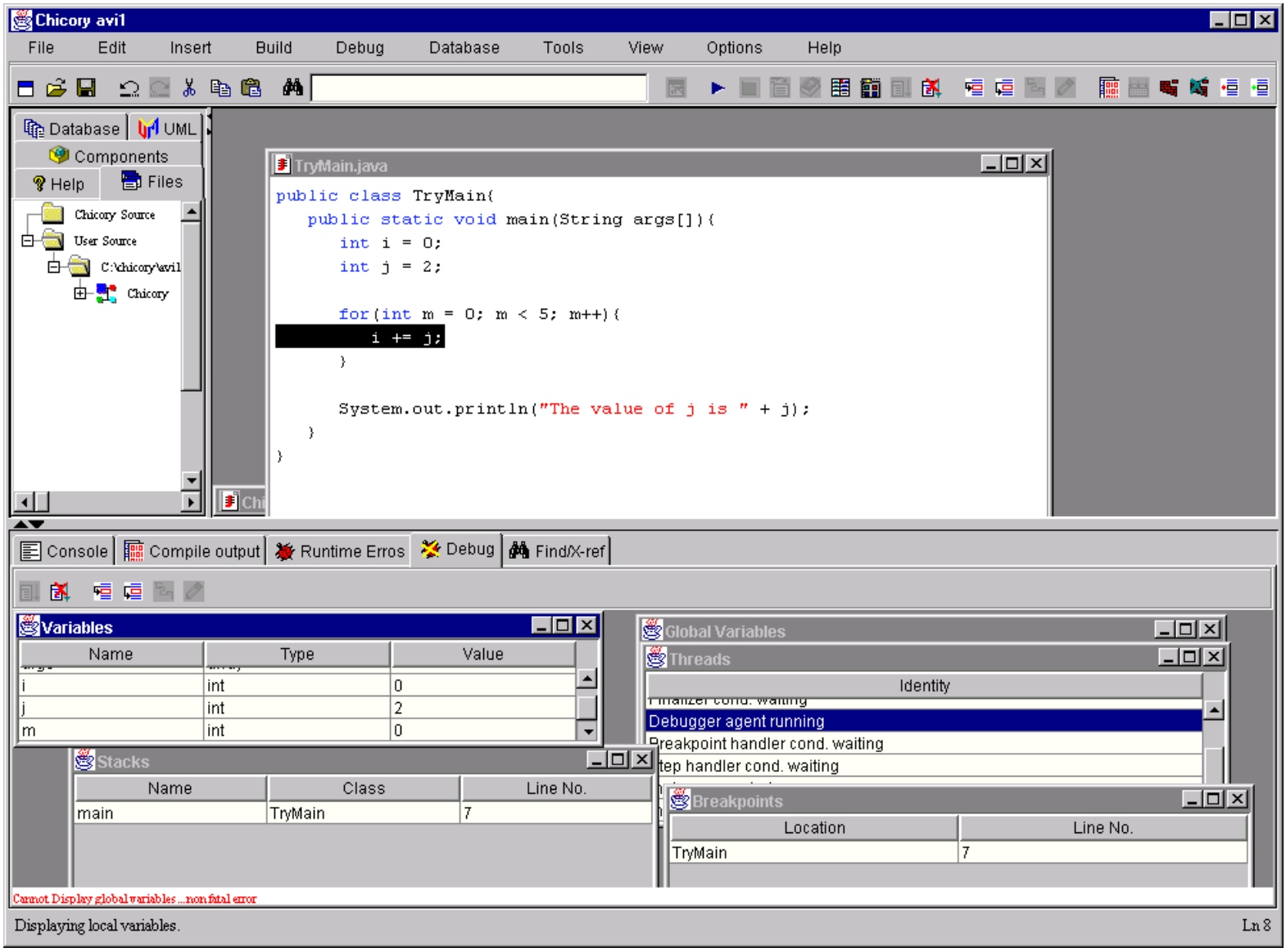
i) The Text Editor



ii) The UML functionality.



iii) The Debugger.



Bibliography

[1] Ashton Hobbs, Tech yourself database programming in JDBC in 21 days,Techmedia,1997

[2] Dan Brookshier, Java beans developer reference, Newriders, 1997.

[3] J.B Rosenberg, How debuggers work, Wiley, 1996.

[4] Martin Fowler, Kendall Scott, UML distilled Applying the standard modeling language, Addison Wesley, 1998

[5] Alex McManus, The need for speed, optimizing your Java programs, Sigs, Java Report, 5/98

[6] Alex McManus, The need for speed, Sigs, Java Report 1/98.

[7] T. Meshorer, Undo/Redo function to your Java Apps with Swing, Sigs, Java Report, 6/98

[8] Tim Prinzing, The Swing Text package, The Swing Connection, 1998.

[9] Tim Prinzing, Customizing a text editor, The Swing connection, 1998.

[10] Eric Armstrong, Understanding Containers, The Swing connection, 1998.

[11] Jeff Denkins, The swing 1.0.2 file chooser, The swing connection, 1998.

[12] http://java.sun.com/products/jdk/1.2/docs/guids/extensions/spec.html , Support for Extensions and Applications in version 1.2 of the Java platform, 1998

[13] Java documentation, Printing, 1998.

[14] Mary Campione & Kathy Walrath, The Java tutorial, Addison Wesley, 1998.

[15] J Gosling, B Joy, G Steele, The Java language specification, Addison Wesley, 1997.

[16] T. Lindholm & F. Yellin, The Java Virtual machine specification, Addison Wesley, 1997

[17] Gamma et al, Design patterns elements of reusable object oriented software, Addison Wesley, 1994

[18] M. Wutka, Hacking Java: The Java professional's resource kit, Que, 1996.

[19] J. Javorski, Java developer's guide, www.mcp.com.

[20] G. Vanderburg et al, Tricks of the Java programming gurus, Sams, 1996.

[21] B. Morgan, Teach yourself sql programming in 21 days, Sams, 1997.

[22] http://www.javasoft.com/products/jdk/1.2/docs/index.html, The Java 1.2 documentation.

[23] http://www.javasoft.com/beans/docs/spec.html, The Java beans specification.

[24] http://java.sun.com/products/jfc/swingdoc-api/overview-summary.html, The Swing Specification.

[25] Lei Xu, MS Thesis, 1997.

[26] Pressman, Software engineering a practitioners approach, McGraw-Hill,1998.

[27] Jar Guide, Java Documentation http://www.javasoft.com/products/jdk/1.2/docs/guide/jar/jarGuide.html.