THE COST ELEMENT BUILDER – A TOOL FOR CREATING AND EDITING SPECIFIC JAVA CODE THROUGH A GRAPHICAL USER INTERFACE

A thesis presented to
the faculty of the
Fritz J. and Dolores H. Russ College of Engineering and Technology
of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Anshuman Taneja
June 2004
This thesis entitled

THE COST ELEMENT BUILDER – A TOOL FOR CREATING AND EDITING
SPECIFIC JAVA CODE THROUGH A GRAPHICAL USER INTERFACE

BY

ANSHUMAN TANEJA

has been approved

for the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Robert P. Judd

Professor, School of Electrical Engineering and Computer Science

R. Dennis Irwin

Dean, Fritz J. and Dolores H. Russ

College of Engineering and Technology
Ohio University, as one of the project members of an industrial project called FIPER, has developed a software tool in Java called the ‘Cost Estimator’, which provides a powerful mechanism for estimating the cost of manufacturing any product. Information representing the product must be written in Java code and plugged into the Cost Estimator to deliver an estimate of the cost.

This work – The Cost Element Builder – is an easy way for a user to be able to write and edit Java code representing his product, through a Graphical User Interface (GUI), without necessarily being adept at Java programming.

The tool is integrated into JBuilder as an open tool. It provides a classical IDE based code generator interface, and provides two-way synchronization between the source code and its graphical representation. The tool aims to find utility for both, novice Java programmers, as well as experienced ones.

Approved: Robert P. Judd,

Professor, School of Electrical Engineering and Computer Science.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>List of Figures</td>
<td>8</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>11</td>
</tr>
<tr>
<td>1.1 The Cost Estimator</td>
<td>12</td>
</tr>
<tr>
<td>1.1.1 Features of the Cost Estimator</td>
<td>13</td>
</tr>
<tr>
<td>1.1.2 Elements</td>
<td>14</td>
</tr>
<tr>
<td>1.1.3 Design and Functionality of the Cost Estimator</td>
<td>16</td>
</tr>
<tr>
<td>1.2 The Need and the Goals of the Cost Element Builder</td>
<td>20</td>
</tr>
<tr>
<td>1.3 Scope of the Builder</td>
<td>22</td>
</tr>
<tr>
<td>1.4 Initial Design Considerations</td>
<td>23</td>
</tr>
<tr>
<td>2. INTEGRATED DEVELOPMENT ENVIRONMENTS AND OTHER JAVA CODE GENERATING TOOLS</td>
<td>25</td>
</tr>
<tr>
<td>2.1 IDE – an Integrated Development Environment</td>
<td>25</td>
</tr>
<tr>
<td>2.1.1 General Features</td>
<td>26</td>
</tr>
<tr>
<td>2.1.2 Editor Features</td>
<td>27</td>
</tr>
<tr>
<td>2.1.3 JBuilder – the chosen IDE</td>
<td>29</td>
</tr>
<tr>
<td>2.2 Other Java code generating tools</td>
<td>30</td>
</tr>
<tr>
<td>2.2.1 Visaj – Visual Application Builder for Java</td>
<td>30</td>
</tr>
<tr>
<td>2.2.2 Arquemie™ for Application Shells</td>
<td>34</td>
</tr>
<tr>
<td>2.2.3 SpeedJG</td>
<td>35</td>
</tr>
</tbody>
</table>
2.2.4 JBuilder Visual Design Tools

2.2.5 Influence on the design of the Builder

3. TOOLS AND API'S SUPPORTING THE BUILDER

3.1 JBuilder

3.2 JOT

3.3 SWING

4. IDEOLOGY AND FUNCTIONING OF THE BUILDER

4.1 The Builder Interface

4.1.1 The Builder Tab

4.1.2 The Structure Pane and the Element Tree

4.1.3 The Properties Table

4.1.4 The Builder ToolBar

4.1.5 The Embedded Cost Estimator

4.2 Creating new Elements and new Components

4.2.1 The New Element Wizard

4.2.2 Buttons on the ToolBar

4.2.3 The New Attribute Wizard

4.2.4 The New Bucket Wizard

4.2.5 The New Method Wizard

4.2.6 The New Map Wizard

4.2.7 Tables and the new Table Wizard

4.2.8 Warnings and the new Warning Wizard
4.2.9 Constants and the new Constant Wizard

4.2.10 The Documentation Generator

4.2.11 The Documentation Links Generator

4.3 Modifying Existing Elements

5. APPLICATION DESIGN AND IMPLEMENTATION

5.1 Design Considerations

5.2 Back End Architecture

5.2.1 ElementViewer and ElementViewerFactory

5.2.2 ElementData

5.2.3 BuilderData

5.3 GUI Architecture

5.3.1 MainPanel

5.3.2 Dialogs

5.3.3 Editing in the Properties Table

5.4 Summary of the Primary Functional Structure

6. FURTHER WORK AND CONCLUSIONS

6.1 Further Work

6.2 Conclusions

REFERENCES

APPENDIX – CREATING AN ELEMENT USING THE BUILDER – AN ILLUSTRATION

A.1 Steps to Building an Element
A.2 Creating the Arm Element 92
A.3 Testing the Arm Element by using the Embedded Cost Estimator 108
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The importance of early Cost Estimation</td>
<td>12</td>
</tr>
<tr>
<td>1.2 Element Structure</td>
<td>15</td>
</tr>
<tr>
<td>1.3 The Cost Estimator Interface</td>
<td>18</td>
</tr>
<tr>
<td>1.4 The Builder in the Cost Estimator Architecture</td>
<td>21</td>
</tr>
<tr>
<td>2.1 The Visage Class Editor</td>
<td>32</td>
</tr>
<tr>
<td>2.2 The Visage Properties Editor</td>
<td>33</td>
</tr>
<tr>
<td>2.3 Arquemie™ Application type selection dialog</td>
<td>34</td>
</tr>
<tr>
<td>2.4 SpeedJG Interface</td>
<td>35</td>
</tr>
<tr>
<td>2.5 SpeedJG Editor</td>
<td>36</td>
</tr>
<tr>
<td>2.6 JBuilder Designer Interface</td>
<td>38</td>
</tr>
<tr>
<td>3.1 The AppBrowser Interface of JBuilder</td>
<td>41</td>
</tr>
<tr>
<td>3.2 SWING Light Weight Component Hierarchy</td>
<td>45</td>
</tr>
<tr>
<td>3.3 Swing Window Classes Hierarchy</td>
<td>46</td>
</tr>
<tr>
<td>4.1 The Builder Interface</td>
<td>48</td>
</tr>
<tr>
<td>4.2 The New Element Wizard</td>
<td>52</td>
</tr>
<tr>
<td>4.3 Buttons on the ToolBar</td>
<td>53</td>
</tr>
<tr>
<td>4.4 The New Attribute Wizard</td>
<td>54</td>
</tr>
<tr>
<td>4.5 The New Bucket Wizard</td>
<td>56</td>
</tr>
<tr>
<td>4.6 The New Method Wizard – Page 1</td>
<td>57</td>
</tr>
<tr>
<td>4.7 The New Method Wizard – Page 2</td>
<td>58</td>
</tr>
</tbody>
</table>
4.8 The New Map Wizard

4.9 The Mapping Dialog

4.10 The New Table Wizard

4.11 The New Warning Wizard

4.12 The New Constant Wizard

4.13 Documentation generated for an Element

4.14 The Documentation Links Generator

4.15 Properties Table editing the Name

4.16 Properties Table editing the Description

4.17 Properties Table editing the Access

5.1 BuilderData Hierarchy

5.2 Dialog Classes in the Builder

5.3 Summary of Primary Functional Structure

A.1 The Arm and its Attributes

A.2 Sub-Elements of the Arm

A.3 JBuilder Dialog to select the type of the New Object

A.4 The New Element Wizard for the Arm

A.5 Shell of the Arm rendered in the Builder

A.6 New Attribute Wizard declaring Length for Arm

A.7 New Attribute Wizard declaring Width for Arm

A.8 New Attribute Wizard declaring Height for Arm

A.9 New Method Wizard (page 1) declaring a Method for the Arm
A.10 New Method Wizard (page 2) declaring a Method for the Arm
A.11 The Builder rendering the Arm and showing the properties of the selected Aggregate Method
A.12 The Added Sub Elements Dialog before adding Sub-Elements
A.13 New Sub-Element Wizard creating an ArmRest
A.14 New Sub-Element Wizard creating an ArmSupport
A.15 The Added Sub Elements Dialog after adding Sub-Elements
A.16 Dialog to set the Map for a Sub-Element
A.17 New Map Wizard creating Map for ArmRest
A.18 Dialog to Define Attribute Associations from the Arm to ArmRest
A.19 New Map Wizard creating Map for ArmSupport
A.20 Dialog to Define Attribute Associations from the Arm to ArmSupport
A.21 Builder Rendering the completed Arm, while executing the Embedded Cost Estimator
CHAPTER 1
INTRODUCTION

The Federated Intelligent Product Environment (FIPER) is a four-year, $21.5 million project initiated with the goal of increasing the profitability of the manufacturing industry, by developing next generation product design and analysis technologies.

FIPER establishes a web-enabled environment that streamlines the design of highly engineered products, by integrating legacy and best-of-breed design and analysis tools. The National Institute of Standards and Technology (NIST) Advanced Technology Program, funds FIPER. Project partners include General Electric, Engineous Software, Goodrich, Parker-Hannifin, Ohio Aerospace Institute, Ohio University and Stanford University [9].

The Ohio University team has developed two application components of the FIPER machine. One is the Data Exchanger – a tool that allows data to be read and translated between various standard industry formats. The other is the Cost Estimator – a tool that allows a manufacturer to estimate and model the cost of manufacturing a product. The Cost Estimator is coded in Java using Java Development Kit (jdk) 1.3.

To estimate the cost of manufacturing a product, it must be represented in Java code conforming to certain templates defined by the Cost Estimator. These products, or even processes for that matter, are known as “Elements”. Hence the initial effort in generating estimates for a product family, involves creating the necessary Elements. This is done by an Element developer, who needs to have knowledge of the Element Java
templates to conform to, as well as the costing algorithms and considerations. Users of the Cost Estimator need not have developed the Elements themselves. A working knowledge of the Cost Estimator and an Element family is what a Cost Estimator user needs at the minimum. The Cost Estimator operates on the compiled “.class” file of an Element to generate a cost estimate for it.

1.1 The Cost Estimator

Cost estimation is a very important factor in the manufacture of a product and must be done early in the product development cycle to have impact. The graph in Figure 1.1 shows why [5].

![Figure 1.1 The importance of early Cost Estimation](image)
Along the X-axis are various stages of product development, beginning with the drawing board and finishing at customer support. ‘Cost Incurred’ is the money spent on the product and predictably it is negligible at the initial stages and increases as the product goes through the stages of design, development, production and finally marketing. The ‘Cost Committed’ is the percentage of the total cost that will be incurred, and at any given stage, it is calculated based on the decisions that have been taken up to that time. As is apparent from the graph in Figure 1.1, these decisions must be made early to have the most impact on cost. Therefore it is imperative to have good cost estimates early in the designs so that informed decisions can be made to reduce the cost.

1.1.1 Features of the Cost Estimator

The Cost Estimator is a tool that allows estimation and modeling of the cost involved in manufacturing a product. The underlying enumeration lists its features.

1. It is a tool that is easy to use early in manufacturing design stages, to obtain costs with little information.
2. It comes with a rich library of Elements representing standard processes.
3. It allows extensibility for adding new proprietary processes and creating product templates with associations.
4. It can develop cost with varying degrees of fidelity, i.e. it can use different costing methods with different algorithms to calculate the cost of the same part. Thus it can be used at different times in the design life cycle.
5. It can be connected to CAD systems through FIPER.
6. It can perform and store trade studies.

7. It can examine the variability of estimates with respect to uncertainties in the parameters.

8. It can calibrate the cost estimate of a new part based on the difference in the actual and estimated cost of a known part.

### 1.1.2 Elements

A brief description of the components of an Element must precede a description of the interface and functionality of the Cost Estimator. An Element is a java class that represents information required to estimate the cost of a part or process. It comprises of the following primary components.

1. **Attributes**: These are the inputs to an Element. For example a wood piece might have the Attributes – length, width, depth, material etc. All Attributes are represented as Java variables in the code for an Element.

2. **Buckets**: These hold calculated values like total cost, labor hours etc. They are represented as Java variables in an Element.

3. **Methods**: These hold the algorithms that do the cost estimation. They are represented as inner classes in an Element. An Element can have multiple Methods. There are two categories of Methods. The Aggregate Method type involves computing the cost of an Element by calculating the cost of its Sub Elements and then aggregating their costs. The Base Method type involves applying an algorithm to the Attributes of an Element to calculate the cost.
4. Sub Elements: The Element can be considered to be constituted of Sub Elements. Sub Elements can include parts and processes and are Elements themselves. For example, a chair could have Sub Elements like Back, Front, Assembly etc. An Aggregate Method uses Sub Elements.

5. Maps: These define value relations between the Attributes of an Element and the Attributes of its Sub Elements, if such relations exist. They are represented as inner classes in an Element.

An example showing the relationships between the components of an Element is shown in figure 1.2 [5].

![Figure 1.2 Element Structure](image-url)
Consider the example of a chair. A chair has several Attributes like its height, material etc. It can have several Buckets that accumulate several types of costs like the material costs, labor hours, and the total cost that may be simply the summation of the material and labor costs. The Estimator offers two types of Methods for cost estimation, that is the BaseMethod, and the AggregateMethod, one of which should be the parent of every Method in an Element. The BaseMethod does not support Sub-Elements, and must employ an algorithm that uses the Attributes of the Element, to determine the cost. The other Method is the AggregateMethod, which allows the usage of Sub-Elements under the Element. While using an AggregateMethod, the Bucket values are calculated by an aggregation of the Buckets of Sub-Elements. The Maps allow mapping between the Element and its Sub-Elements. Say for example, a chair has a back as a Sub-Element, and if the height of the back is desired to be the same as that of the chair, then this can be mapped using the Maps.

1.1.3 Design and Functionality of the Cost Estimator

The Cost Estimator is a stand-alone Java application. The front-end is created with components from the Java package - SWING. Its display primarily comprises of the following components.

1. Features or attributes of the Element being cost estimated.

2. The sub-elements of the main Element, if any.

3. A listing of all the costs, including a break up of how each sub-element contributed to it.
4. The various methods or algorithms used by the Element for costing.

Figure 1.3 shows the interface of the Cost Estimator estimating the cost of manufacturing a chair.
Figure 1.3 The Cost Estimator Interface
An examination of figure 1.3 reveals the following. The pane on the left displays a tree like representation of the current Element and its Sub Elements. Beside each Sub Element is an estimate of what it contributes to the cost.

There are tabs associated with each different costing Method present in the Element. Each Method has a listing of Attributes displayed along with their editable values. The user can tweak the values of each attribute and see how it would affect the cost. This is a very valuable feature as can be appreciated by considering the dilemma of a design engineer designing a jet engine, and wondering how would increasing each nozzle length by 2 mm affect the total cost. Using the Cost Estimator, this dilemma is solved nearly as quickly as he can type ‘2’ into an appropriate field and hit return.

The Summary Table displays the various Buckets along with their values for all the Sub Elements if applicable. The Pie Chart graphically displays the contribution of each Sub Element to the total cost. There are tabs for Risk Analysis and Trade Studies, clicking on which fire up the respective modules. The user can store the state of an estimate as a Trade Study and reopen it at a later time to resume work. Multiple Trade Studies can be stored and compared against each other. Trade Studies are stored as XML files. The Risk Analysis module can examine the variability of estimates with respect to uncertainties in the parameters. There is a Calibration module as well, which is not displayed in figure 1.3. It can calibrate the cost estimate of a new part based on the difference in the actual and estimated cost of a known part.
1.2 The Need and the Goals of the Cost Element Builder

Not all manufacturing engineers are adept at Java programming. The Element developers need to write Java code to represent Elements, and often Cost Estimator users too need to write some Java code to specialize the Elements to their needs. To aid these situations, the Ohio University FIPER project team developed the Cost Element Builder. This is a tool that allows a user to easily generate and edit Java code, specific to the needs of the Cost Estimator, through a GUI.

The Cost Element Builder will be referred to as just the ‘Builder’ from this point forward in this text. Figure 1.4 shows where the Builder, (marked as Element Builder), fits into the architecture of the Cost Estimator [5]. The Element Library in the figure refers to a collection of Java files, each of which represents an Element. The Builder is shown to be creating Elements, which get added to this library.
Figure 1.4 The Builder in the Cost Estimator Architecture.

The following points outline the goals for the Builder.

1. It should cut out the laborious work involved in creating Elements.
2. It should reduce significantly the time taken to create Elements.
3. Code generated through the Builder should be guaranteed to be free of compilation errors.
4. It should provide two-way synchronization between the Element Java code and a graphical representation of the Element.

5. It should maintain projects containing several Elements.

6. It should provide a debugging environment.

The Builder was designed to cater to two categories of users of the Cost Estimator and assume a different role for each. The first type of user is the Element developer who needs to be familiar with Java as well as the manufacturing considerations. To this kind of user, the Builder should primarily be a time saving tool. The Builder should rapidly develop the structure of an Element, making this otherwise labor intensive process, convenient and much faster. The second kind of user is someone who uses the Cost Estimator with the Element family already defined, and maybe a Java novice. This user may need to make some modifications to Elements to specialize them to his needs. The Builder should make these modifications possible through the GUI of the Builder instead of manual code modifications.

### 1.3 Scope of the Builder

The Builder is a tool for generating and editing Java code specific to the needs of the Cost Estimator. The emphasis on specific is of great importance here. There is no tool yet, that can generate Java code of every type through a GUI. The sheer vastnesses of Java APIs make such a tool unfeasible.
There are several tools available commercially that can generate and edit Java code through a GUI, and each has its scope. The most common family among these tools is that of GUI designers. These tools help in generating Java code that represents graphical user interfaces.

The Builder can generate and edit Java code that represents Elements only. The Builder does not recognize code that does not represent an Element and does not try and parse it.

1.4 Initial Design Considerations

There were two initial design options for the Builder.

1. A stand-alone Java application.

2. An application integrated with an existing Java integrated development environment (IDE).

A stand-alone application’s biggest advantage was that its development was entirely independent of any external influences. An application integrated with an existing IDE suffered on this account because of the following reasons.

1. Its existence would depend on the existence of the IDE.

2. Bugs, if any, encountered in the IDE, would need to be worked around, and in many cases, the source code for the IDE is not available.

It was realized early in the development process, that many of the features required by the Builder, already existed in several Java IDEs. Instead of “reinventing the wheel”,
it was considered easier to augment an existing IDE with Builder unique functionalities rather than recreating IDE functionalities in the Builder. Thus the decision to integrate the Builder with a Java IDE was taken by the project team.
CHAPTER 2
INTEGRATED DEVELOPMENT ENVIRONMENTS AND OTHER JAVA CODE GENERATING TOOLS

Since the Builder is an IDE in its functionality, this chapter will first take a look at what the desirable features of an IDE are. This chapter will also examine some of the existing commercial tools that do Java code generation through a GUI, and how they influenced the design of the Builder.

2.1 IDE – an Integrated Development Environment

An IDE is generally a programming environment that has been packaged as an application program, typically consisting of a code editor, compiler, debugger, and GUI designer [12].

1. Code Editor: This is a text editor where the user can view and edit source code. In its most simplistic form, it is nothing more than a text area that responds to keystrokes. Generally a good code editor has many more features.

2. Compiler: This allows the user to compile code and results in errors, if any.

3. Debugger: This allows the user to debug errors in the code.

4. GUI builder: This is a tool that allows the user to create and edit source code through a GUI.
Before the use of IDEs became widespread, programmers had to make do with
difficult to use command line compilers, linkers and loaders, code debuggers, logic
analyzers, and other support tools that required an extensive knowledge of both the tools
and the underlying hardware. Moreover, these tools were all run individually from the
command line, without the ability to be used interactively.

The IDE accomplishes two major things for programmers. First, it enables them
to work with multiple tools simultaneously, without having to exit one in order to launch
another. This improves productivity and code quality, since it’s easy to refer back and
forth between displays and outputs. Second, IDEs provide a common appearance and
behavior between tools [11].

Requirements for IDEs are now much more demanding than in the past. Initially,
IDEs typically consisted of just the first three features mentioned above. The current
IDEs have the same core tools, but even the simplest IDE exhibits numerous
enhancements that reflect today’s development environment. This growth in IDE
complexity has resulted from the increased complexity of application environments [13].

Naturally, the Builder would be expected to provide as many desirable IDE
features as possible. The following listing provides in greater detail, features that the
Builder being a good IDE, should provide.

2.1.1 General Features

1. The ability to organize source files into projects and to display them accordingly.

For example, displaying the files and packages in a project in a tree structure.
2. The ability to display multiple documents at the same time, with perhaps a tabbed document view.

3. The ability to open and save files. The ability to create new ones too.

4. The ability to compile and run at the hit of a button.

5. Complete two-way synchronization between the source code and the GUI builder objects.

6. The ability to navigate using just the keyboard.

7. Display of class hierarchies in a project.

8. The ability to add tools or plug-ins.

9. It should be visually appealing.

2.1.2 Editor Features

1. The ability to color and highlight syntax. With this, it is easy to tell keywords, comments and plain text apart.

2. Cut and paste

3. Search and replace.

4. Font functionalities.

5. Line numbering.

6. Undo/Redo capabilities.

7. Formatting and indentations.

8. The ability to automatically complete half typed out words.
9. The ability to detect when a file has been modified externally, and take appropriate action.

In addition to the above, Java IDEs should have features like classpath management, the ability to add and edit libraries to the build path, and runtime configuration management. If all the features above were to be implemented from scratch for the Builder, even by conservative estimates it would take several years to develop. By integrating the Builder with an existing commercial IDE, most of the features mentioned above, would be provided at the onset.

The following are two architectural options IDEs have [14].

1. Tightly integrated/closed system:

   In this method, the mode of integration requires that standards for data transfer and control between tools be established. This approach finds widespread application in commercial editor-based IDEs where the editor, compiler and assembler are tightly integrated. Generally, it is difficult for users to add an additional tool to such a system.

2. Loosely integrated/ open system:

   In this architecture, the integration involves tools communicating via a controller. This controller is central to the IDE. If a tool wishes to talk to another, then it must do so via the controller.
2.1.3 JBuilder – the chosen IDE

The Borland IDE for Java, JBuilder, is of the open type and adding a new tool to it is not difficult. It provides many of the features that we desired the Builder to provide. If JBuilder was chosen as the IDE to integrate the Builder with, the only feature to be implemented would be a specialized GUI builder (Builder), which would be in addition to the GUI Swing designer provided by JBuilder, and a two-way synchronization between it and the source code.

The other IDE examined and considered by the project team was IBM’s Eclipse. After careful analysis, JBuilder was considered better suited to the development of the Builder than Eclipse. The primary reasons were the following.

1. Eclipse was a relatively new product in the market as opposed to JBuilder, which had been in the market for several years.

2. The documentation for open tools in JBuilder was considered more descriptive and complete by the project team.

3. JBuilder primarily utilized SWING for its GUI components, as opposed to a standard widget kit used by Eclipse. Since the plan was to use SWING for the GUI components of the Builder, it was less likely to have compatibility issues with JBuilder.

With the above in mind, the decision to integrate the Builder with JBuilder was taken.
2.2 Other Java code generating tools

With the decision to integrate the Builder with JBuilder being made, the next consideration was the interface design of the Builder. Several tools exist in the market, which perform a function similar to the Builder – Java code generation through a GUI. However, none is specialized to do what the Builder does – build Elements. Nevertheless, the interface design paradigms in code generation tools could be of use in the interface design of the Builder. Hence some such tools were examined to determine the interface design of the Builder. This section will take a look at these tools.

2.2.1 Visaj – Visual Application Builder for Java

Visaj is a Java development tool, which allows graphical development of the structure and interface of an application. When the graphical representation is complete, the developer can generate Java code. Not only does Visaj generate Java code, it is also written in Java. Both the generated code and Visaj itself are portable across all platforms that support Java 1.1 and above. Visaj is an Imperial Software Technology (IST) product. The company is London based and is one of the leaders in software engineering tools used to construct graphical user interfaces for application software [7].

Creating Java applications and applets becomes a simple point-and-click exercise with Visaj. Following are some of the features this tool provides [8].
1. A dynamic display of the user interface being designed. As one is putting objects together, one can lay them out and set their resources. They will be running in a live window exactly as they will be when code is generated.

2. A palette of Java objects - by default the abstract widget toolkit (AWT), SWING and IST Diamonds. One can click on an object to add it to a design and working window. The palette can be configured to add customized or third party objects.

3. Easy to use layout editors for all the Java layouts. One can drag and drop objects and specify constraints.

4. Full class editing. One can add and delete methods, change the class and method signatures, set up constructors and a main method.

5. Pure Java code is generated from a design. Pure Java means platform independence.

Figure 2.1 is a screenshot from the Class Editor of this tool. The Class Editor is where the substance of the user interface is defined and source code generated. Apart from designing and editing class files, buttons on the toolbar and in the menus, lead to the project window for organizing the files within a project [8].
Figure 2.1 The Visage Class Editor

Figure 2.2 is a screenshot from the Property Editor of this tool. The Property Editor is used to edit the properties of any component of the code. In Figure 2.2 it is editing the properties of a JButton. This dialog is pretty exhaustive for most components and provides a powerful editing and customizing mechanism [8].
Figure 2.2 The Visage Properties Editor
2.2.2 Arquemie™ for Application Shells

Arquemie™ for Application Shells is a Java Development tool, in the form of an IDE plug-in. It automates the generation of Java SWING source code for only two types of applications - user interfaces implementing a Standard MDI (Multiple Document Interface) Desktop or a Tabbed MDI Desktop style. It is a Xintegra Technologies™ product. They provide Java development tools and have a suite of IDE plugable tools for automating laborious coding activities [9].

This is a specialized GUI developer, and provides the user the ability to customize his GUI application to fine detail. Figure 2.3 shows one of the dialogs from this tool. This dialog decides which of the two types of applications will be pursued [9].

![Figure 2.3 Arquemie™ Application type selection dialog](image)
2.2.3 **SpeedJG**

SpeedJG is an XML based GUI builder tool for creating Java SWING applications. The core part of this tool is a parser that reads the meta-data described in XML to create Java GUI components on the fly. An IDE enables the Java developer to design GUIs, generate their meta-data, check their layout, and create the corresponding source code [10].

This tool supports a fair number of Swing components. Figure 2.4 shows its interface [10].

![Figure 2.4 SpeedJG Interface](image-url)
The working panels are located in the center of the GUI and are used to navigate, to edit and to see the results of the designing activities. On the left-hand side is a tree showing all the components the projects and GUIs consist of. As the structure of a Java GUI is built up from different components containing other components and so on, this structure optimally fits the structure of XML. Therefore all the meta-data of the GUI project, i.e. the description of the components and their properties, are stored as XML and presented within this tree. All the menu and toolbar items found on the SpeedJG screen correspond to the currently selected component within this tree, and within the context of the tree selection these menu and toolbar items are enabled or disabled.

The Editors Panel consists of different tabs showing the components currently being worked on. Figure 2.5 shows the Editor Panel [10].

![Figure 2.5 SpeedJG Editor](image)
The properties of the components are defined here and these properties are customized with respect to the component currently being edited. For example, a JFrame editor only shows those properties that make sense for a JFrame and not all the Component and Container properties a JFrame inherits. The properties can be edited using this dialog.

2.2.4 JBuilder Visual Design Tools

JBuilder provides tools for generating and editing code specific to GUIs. These tools are called the visual design tools and they include a UI designer, a menu designer, and a column designer. Each of these tools consists of a component palette, an Inspector, several designers, and a component tree.

The UI designer is used in assembling visual UI components that inherit java.awt.Container. When the Design tab is first clicked after opening a file in the content pane, JBuilder displays the UI designer by default. UI components appear in the UI folder of the component tree. The menu designer is used to create menus. Menu components appear in the Menu folder of the component tree. The column designer allows a user to work visually with data set components. Data set components appear in the Data Access folder of the component tree. Figure 2.6 shows the JBuilder Designer Interface [2].
Figure 2.6 JBuilder Designer Interface.

The component tree displays the components of the GUI in a hierarchical structure. Selecting any component brings up its details in the Inspector. The user can make changes to the values of the component through the Inspector. New components can be added by selecting them from the component palette.
2.2.5 Influence on the design of the Builder

Inspection of these tools revealed that there are some features common to all of them. These features were incorporated into the design of the Builder and are listed below.

1. A structure representing the various components of the Java class under construction.
2. A specialized dialog or table to edit one component at a time.

The eventual layout of the Builder was influenced the most by the layout of the JBuilder Designer Interface, as will be apparent later in this text.
CHAPTER 3
TOOLS AND API'S SUPPORTING THE BUILDER

As in any software, there are various tools and programming packages that hold the Builder together. At the front end there is JBuilder providing its IDE, and the javax.swing package of API’s providing all of the other GUI components. Collectively these two create the interface of the Builder. At the back end, JBuilder provides the APIs powering most of the work, for use with its open tools. The primary package is com.borland.jbuilder.jot, or just “JOT”. This package provides the infrastructure necessary to read, alter, and regenerate source code.

3.1 JBuilder

JBuilder is a Java IDE created by Borland, and is one of the leading cross-platform environments for building Java applications. Figure 3.1 shows the layout of the interface of JBuilder, also known as the “AppBrowser” [2].
Figure 3.1 The AppBrowser Interface of JBuilder
Of the features marked in figure 3.1, the following have significant relevance to the Builder:

1. Project Pane: This component shows the structure of a Java project. It displays all the files in the project in a tree structure. This component is not open for modification. The node or file selected in this pane is the active node or file.
2. Structure Pane: This provides a summary of the active file by parsing its contents.
3. Content Pane: This displays the contents of the active file, which is the source code in case the active file is a Java source file.
4. File View Tab: The tabs that appear at the bottom of the Content Pane depend on what kind of file is open. Only tabs appropriate to the open file appear here. Each of these tabs provides a different view of the open file.

JBuilder is designed to be an extremely open, extensible product. It provides the Open Tools application programming interface, which contains the classes and interfaces a user needs to create his own Open Tools to extend JBuilder. The following section takes a look at the Open Tool package the Builder uses most extensively – JOT.

3.2 JOT

JOT stands for the Java Object Toolkit and is a set of interfaces and classes used to parse and generate Java code. Its main function is the parsing of source or class files to extract information about their contents. Additionally, it can be used to generate code. The heart of JOT is a series of interfaces that describe the Java language in detail.
JOT can parse Java classes and Java source files. The methods specified by JOT for parsing Java use the naming convention get<XYZ> to extract whatever XYZ represents. These getter methods are generally used to parse the nested nature of Java code and usually return another item that is represented by a JOT object. While parsing from Java source, one can extract a little more information than from a Java class. This is because some items represented in the source (such as comments) are stripped out at compile time and aren't represented within the class file. During parsing, everything works from the outside in. At the outermost level (a JotSourceFile), the parsing procedure is simple. A JotSourceFile contains one or more classes (represented by a JotClass or a JotClassSource), and each of those classes is composed of variables and methods. Usually a loop is used to iterate through all the classes in the file. From each of those classes an array of methods defined in that class can be retrieved and one can loop through each of these. The implementation of these methods makes things a bit more complicated. For example, a class can have many methods and each method can be composed of several statements, and each of those statements can be a method that is composed of many statements, and so on. At this level, recursion is common. JOT allows parsing down to the atomic level.

JOT can be used to write source as well. When writing source, the programmer deals with much higher-level items like methods or statements instead of the very fine-grained items available when reading source or class files with JOT. The methods specified by JOT for writing Java code use the naming convention add<XYZ> and set<XYZ> where XYZ represents the item being created or modified. JOT is unforgiving
of being told to write bad code and will throw a JotParseException once it has tried to read back anything the programmer has attempted to write with syntax errors in it [2].

3.3 SWING

The package javax.swing provides a set of "lightweight" components that, to the maximum degree possible, work the same on all platforms [3]. Swing provides many standard GUI components such as buttons, lists, menus, and text areas, which can be combined to create an application's GUI. It also includes containers such as windows and tool bars. In general, the class names that start with "J" are the components added to a graphic layout. Examples: JButton, JDialog, JLabel, JList, JPanel, JTable, JTree. The remaining files in the swing package contain the utility classes and interfaces that the components use to function. Figure 3.2 shows the hierarchical structure of some of the most common lightweight components provided by SWING [3]. These are also known as Widgets. Figure 3.3 shows the Window hierarchy in SWING [3].
Figure 3.2 SWING Light Weight Component Hierarchy.
Figure 3.3 Swing Window Classes Hierarchy.

The Builder extensively uses SWING to create GUI components that the user interacts with.
CHAPTER 4
IDEOLOGY AND FUNCTIONING OF THE BUILDER

As mentioned earlier, the Cost Estimator generates estimates for Elements, which are products or processes represented in Java. The Builder is a tool that can create and edit the Java code that represents these Elements. The Element developer needs to conceptualize the features of his Element and the costing algorithm he wishes to use, and then enter this information into the Builder using its myriad wizards and tables. The Builder subsequently writes out Java code into a file, which accurately represents the conceptualization. Most of the code can be edited through a GUI. A look at the Builder interface follows.

4.1 The Builder Interface

The figure 4.1 shows the interface of the Builder. Comparing this to figure 3.1 - The JBuilder Interface - one can see what has been retained from JBuilder and what has been added.
Figure 4.1 The Builder Interface
4.1.1 **The Builder Tab**

In figure 3.1 it is seen that JBuilder provides 5 tabs. In figure 4.1, a new tab can be seen – the Builder Tab. Clicking on this tab fires up the Builder. In typical usage mode, the user shuttles between the Builder Tab and the source tab. Clicking on the source tab displays the source code for an Element, and clicking on the Builder Tab displays the Builder’s representation of the Element.

4.1.2 **The Structure Pane and the Element Tree**

The Structure Pane contains the Element Tree. The Element Tree is a representation of the components of an Element in a tree like structure. The nodes of the tree are the chief component types like Attributes, Buckets, Methods and Maps. The child nodes of these, or the leaves, are the various Attributes, Buckets, Methods and Maps present in the Element. There are a few more node types listed under the Element Tree in figure 4.1, which will find mention later in this text.

The user can select any leaf to bring up the details of that leaf in the Properties Table (far right corner in figure 4.1). The Element Tree is a drag source, which means that the user can drag objects off the tree to drop them at appropriate locations. This finds utility in selecting the Attributes that go into the calculations of certain Methods.

4.1.3 **The Properties Table**

This table reflects the details of the currently selected leaf in the Element Tree. Its left column lists the various properties of the leaf and its right column lists the
corresponding values. In cases where the value is too complex to list in a single row, “…” is displayed. Clicking on such rows brings up a specialized dialog, which displays the value.

The Properties Table provides the editing mechanism in the Builder. Any changes to be made to an Element by means of the GUI, have to be made through this table. In some cases, where the property is a simple string or number, the editing can be done right in the cell where it is displayed. In other cases, a specialized dialog, works as the editing tool. All changes made through the Properties Table reflect immediately in the source code of the Element.

4.1.4 The Builder ToolBar

The Builder ToolBar contains all the buttons that bring up wizards to create new Attributes, Buckets, Methods, Maps and more.

4.1.5 The Embedded Cost Estimator

Also known as the Builder Runner, it is essentially the Cost Estimator embedded into the Builder. It provides an easy way to test the Element being created or edited, against the tool it is being created for. The Builder Runner generates dynamic cost estimates for an Element. This module was primarily developed by Dr. Wenle Zhang, a member of the project team.
4.2 Creating new Elements and new Components

This section will cover the functioning of the Builder when it creates a new Element. There are several wizards encountered en route. A figure showing each wizard will be followed by a brief description of the same.

4.2.1 The New Element Wizard

This wizard is brought up by clicking the “New” icon on JBuilder’s tool bar or going through the menu bar and clicking on “File” followed by “New”. That brings up a dialog, which displays several options regarding the type of new object desired to be created. Among the choices, “Cost Element” should be selected. That brings up the wizard shown in figure 4.2.
This wizard creates a new Java file, which represents an Element. It also takes care of the package the user would like his Element to be placed in. The imports are the libraries to be included in the code. At the completion of this wizard, a “.java” file is created which contains the imports and constructors for the Element. This is a start for the task of building an Element. The functionality for this wizard was kept at a minimum because it was considered better to have several brief wizards rather than a single lengthy one. The above wizard creates the following lines of code on hitting “Ok”.

```
edu.ohiou.estimator*
edu.ohiou.util*``
package chair;
import edu.ohiou.estimator.*;
import edu.ohiou.util.*;
/**The class NewChair*/
public class NewChair extends Element {
    public NewChair() throws EstimatorException {
        this(null, null);
    }
    public NewChair(Class map) throws EstimatorException {
        this(map, null);
    }
    public NewChair(Class map, String name) throws 
    EstimatorException {
        super(map, name);
    }
}

4.2.2 Buttons on the ToolBar

The ToolBar contains 9 Buttons, most of which bring up a wizard to create a specific type of component of an Element. Figure 4.3 shows what the ToolBar looks like.

![Figure 4.3 Buttons on the ToolBar](image_url)

The toolbar however is not the only place a user can bring up these wizards. There is a facility to right click in the structure pane and bring up any of these.
4.2.3 The New Attribute Wizard

This wizard creates a new Attribute for the Element. Although figure 4.4 shows just one face of this wizard, there are twelve distinct faces it takes on. This is because there are twelve different types of Attributes that can be created using this wizard. Each type has its own requirements and hence this wizard morphs for each type. This is basically a JDialog and is one of the more complex classes in the Builder.

![Create New Attribute](image)

Figure 4.4 The New Attribute Wizard
The entries into this wizard as shown in figure 4.4 would create the following lines of code on hitting “Ok”.

```java
/**this attribute reflects the height of the chair*/
public DoubleAttribute height = new DoubleAttribute(
        this, "Height", 5.7, false, 0.0, 20.0,
        false, 2, Unit.INCH, null);
```

### 4.2.4 The New Bucket Wizard

This wizard creates a new Bucket for the Element. This is one of the simplest wizards because the complexity of a bucket lies not in its definition, but in the assignment of its value. The Builder handles only the definition of the Bucket, while the assignment of its value must be manually done by the user, depending on his algorithm. Figure 4.5 shows the wizard used to create new Buckets.
The entries into the wizard as shown in figure 4.5 create the following lines of code on hitting “Ok”.

```java
/**
 * This holds the summation of the
 * total cost of this chair
 */
public DoubleBucket totalCost = new DoubleBucket(
    this, "TotalCost", Unit.DOLLAR);
```

### 4.2.5 The New Method Wizard

This is the only two-page wizard because Methods are the most detailed components of an Element. It creates a new Method for an Element. A Method is
represented as a Java inner class and it contains the algorithm for estimating the cost. While the algorithm itself is something the user needs to input manually, the Builder creates a good frame for the Method and can even add Attributes and Sub-Elements into a Method for calculation purposes.

The first page (figure 4.6) takes in generic inputs like name and description. The second page (figure 4.7) has a grid to drop Attributes, which have been dragged from the Element Tree using the mouse. These Attributes are displayed by the Method, when the Cost Estimator is executing it. In fact, they will be displayed in the Cost Estimator in the same positions as this grid displays them.

Figure 4.6 The New Method Wizard – Page 1
Figure 4.7 The New Method Wizard – Page 2
The entries shown in figures 4.6 and 4.7 will create the following lines of code.

```java
/**
 * This method is of aggregate type
 * and uses sub elements to
 * calculate cost.
 */
public class ComplexMethod extends AggregateMethod {
    public ComplexMethod() throws EstimatorException {
        super("ComplexMethod", NewChair.this);
        addAttributes(new
            Attribute[]{height, null, parentCount}, 0, 0);
        addAttributes(new
            Attribute[]{null, quantity, null}, 1, 0);
    }

    public void initAttributes() {
    }

    protected void calculate() {
    }
}

4.2.6 The New Map Wizard

This wizard creates a new Map for an Element. A Map is represented as an inner
class and performs the task of mapping the values of certain Attributes of the Element to
the values of certain Attributes of a Sub Element. Figure 4.8 shows what this wizard
looks like.
This wizard brings up another important wizard (figure 4.9) when the button “Attribute Associations” is clicked. That wizard can create simple mappings between Attributes. It can list more complex maps when fired through the Properties Table, but cannot modify them.
The entries as shown in figures 4.8 and 4.9 create the following lines of code.

```java
/**
 * This map associates attribute values from
 * the NewChair to its sub element - Back.
 */
public class NewChairToBackMap extends Map {

    public void initAttributes(Element element) throws EstimatorException {
        Back ele = (Back)element;
        ele.height.setValue(height.getValue());
        ele.parentCount.setValue(parentCount.getValue());
        ele.quantity.setValue(quantity.getValue());
    }

    public String getName() {
        return "ChairBackMap";
    }
}
```
4.2.7 **Tables and the new Table Wizard**

Elements typically use tables of values in calculations involving cost estimation. An example may be a table of material costs or labor rates. “Table” is a class defined by the Cost Estimator that can store values taken from a csv file or an array of Objects (java.lang.Object).

The Builder supports generation of new Tables of both kinds, but can edit only the tables that take their values from Object arrays. Tables are generally declared as static objects because they usually contain information that is invariable with respect to instances of Elements. The wizard that creates new Tables is fairly detailed and is shown by Figure 4.10.
Figure 4.10 The New Table Wizard

The above entries in the wizard produce the following lines of code upon hitting “Ok”.

```java
public static final Table NewTable = new Table( new 
    Object[][]{{"we", new Integer(1), new Double(1.5)},
        {"are", new Integer(2), new Double(2.5)}},
    new String[]{"COL1", "COL2", "COL3"});
```

### 4.2.8 Warnings and the New Warning Wizard

Warnings are components of an Element. They contain within them a message, which is displayed whenever a certain condition concerning user inputs, is violated. For
example, if the user enters the value of an Attribute such as height, that exceeds the acceptable limit, a Warning is posted on the interface of the Cost Estimator. The Builder supports creation and editing of Warnings. Figure 4.11 shows the wizard that creates new Warnings.

![Create New Warning Window](image)

Figure 4.11 The New Warning Wizard

Warnings themselves are objects, which reside in the Element. They do not have any idea of the conditions under which they are posted; they only know what their message is. The above entries produce the following lines of code.
/**
 * This warning is posted when a user
 * enters an Attribute value too large.
 */

Warning newWarning = new Warning(
    this, "NewWarning", "Exceeding upper limit.");

4.2.9 Constants and the New Constant Wizard

Elements typically hold Java variables as constants, when they will be used in more than one place and/or by other related Elements. They are declared as final and static. These constants are of the primitive type as defined by the Java language specifications with the exception of String (java.lang.String). The Builder supports creation and editing of constants. Figure 4.12 shows the wizard used to create a new Constant.

![Figure 4.12 The New Constant Wizard](image-url)
The above entries in the wizard produce the following lines of code.

```java
/**This integer is a constant test.*/
public static final int testConstant = 23;
```

### 4.2.10 The Documentation Generator

The Documentation Generator creates documentation for an Element in the form of an Html file. The file contains a tabulated detailed summary of the contents of the Element. There are two sections to the file. One section is generated from the content in the source code of the Element. The other is a user data section where the user is free to add additional data to describe any component of the Element. The former section has a sub section for each component, and included in that sub section is a link to user-defined data for that particular component, which is in the latter section. Thus the Html file contains all the descriptive information from the source code, and allows additional information to be added, with links to it. This Html file can be edited through the editor provided by JBuilder. This module has been primarily developed by Dr. Wenle Zhang, a member of the project team. Figure 4.13 displays an example of the documentation file created for an Element, as rendered by Microsoft Internet Explorer.
**Element: SystemElement.java**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Material</td>
<td>List</td>
<td></td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>material</td>
<td>List</td>
<td></td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
<td></td>
<td>SystemElement</td>
<td>Inherited from Element</td>
</tr>
<tr>
<td>Quantity</td>
<td>Integer</td>
<td></td>
<td>1</td>
<td>Inherited from Element</td>
</tr>
<tr>
<td>Parent Element</td>
<td>Integer</td>
<td></td>
<td>1</td>
<td>Inherited from Element</td>
</tr>
</tbody>
</table>

**Bucket Summary**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labor</td>
<td>Double</td>
<td>hr</td>
<td></td>
</tr>
<tr>
<td>Material</td>
<td>Double</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>Total Cost</td>
<td>Double</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

**Method Summary**

**Warning Summary**

**User Information**

Figure 4.13 Documentation generated for an Element
4.2.11 The Documentation Links Generator

This is a wizard that is used to select the components of an Element, which need their own user data areas created in the user defined data section of the Html documentation. The user can choose the components to be mentioned in the user defined section, and likewise, choose the components to be ignored in that section.

This wizard displays a tree similar to the Element Tree from the structure pane, in a dialog, with the exception that each leaf has a checkbox attached to it. If the user checks a box, the Element component corresponding to it, gets an individual data area created in the Html documentation. Figure 4.14 shows what this wizard looks like.
Figure 4.14 The Documentation Links Generator
4.3 Modifying Existing Elements

There is only one avenue for modifying an existing Element through the Builder GUI, and that is the Properties Table. As explained earlier, this Table is a two column, multi row table, which displays and can edit the details of a selected leaf in the Element Tree. Figures 4.15 through 4.17 show some editing modes of the Properties Table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Height</td>
</tr>
<tr>
<td>D</td>
<td>height</td>
</tr>
<tr>
<td>Access</td>
<td>protected</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>DoubleAttribute</td>
</tr>
<tr>
<td>Unit</td>
<td>in</td>
</tr>
<tr>
<td>Default Value</td>
<td>0.0</td>
</tr>
<tr>
<td>Must Have?</td>
<td></td>
</tr>
<tr>
<td>Alias Names</td>
<td></td>
</tr>
<tr>
<td>Throws Exception</td>
<td>Soft, Hard</td>
</tr>
<tr>
<td>Lower Limit</td>
<td>0.0</td>
</tr>
<tr>
<td>Upper Limit</td>
<td>70.0</td>
</tr>
<tr>
<td>Precision</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 4.15 Properties Table editing the Name
### Model Number

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Model Number</td>
</tr>
<tr>
<td>ID</td>
<td>model</td>
</tr>
<tr>
<td>Access</td>
<td>public</td>
</tr>
<tr>
<td>Description</td>
<td>...</td>
</tr>
<tr>
<td>Type</td>
<td>StringAttribute</td>
</tr>
<tr>
<td>Unit</td>
<td></td>
</tr>
<tr>
<td>Default Value</td>
<td>null</td>
</tr>
<tr>
<td>Must Have?</td>
<td>✓</td>
</tr>
<tr>
<td>Alias Names</td>
<td>...</td>
</tr>
</tbody>
</table>

![Model Number Table](image)

Figure 4.16 Properties Table editing the Description

### Depth

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Depth</td>
</tr>
<tr>
<td>ID</td>
<td>depth</td>
</tr>
<tr>
<td>Access</td>
<td>unprotected</td>
</tr>
<tr>
<td>Description</td>
<td>public</td>
</tr>
<tr>
<td>Type</td>
<td>protected</td>
</tr>
<tr>
<td>Unit</td>
<td>private</td>
</tr>
<tr>
<td>Default Value</td>
<td>package</td>
</tr>
<tr>
<td>Must Have?</td>
<td></td>
</tr>
<tr>
<td>Alias Names</td>
<td>...</td>
</tr>
<tr>
<td>Throws Exception:</td>
<td>Soft, Hard</td>
</tr>
<tr>
<td>Lower Limit</td>
<td>0.0</td>
</tr>
<tr>
<td>Upper Limit</td>
<td>40.0</td>
</tr>
<tr>
<td>Precision</td>
<td>2</td>
</tr>
</tbody>
</table>

![Depth Table](image)

Figure 4.17 Properties Table editing the Access
CHAPTER 5
APPLICATION DESIGN AND IMPLEMENTATION

The Builder application is spread over 97 Java classes. Understandably, its structure is complex. This chapter will try and explain the design considerations and the final programming implementations of design.

5.1 Design Considerations

The Builder works on building an Element for the purpose of cost estimation. Hence there has to be a class that represents an Element. It needs to hold objects that represent its various components like Attributes, Buckets etc. The next consideration is, where and how, does the Element get represented. The obvious solution is the structure pane. The Element is represented similar to the manner in which JBuilder represents a file in the structure pane.

The details of a specific Component need to be displayed. The approach adopted is to display the details of just one component at a time, when that component is selected. This offers a greater degree of detail in the display, per component.

The target users of the Builder are not necessarily adept at programming. Keeping this in mind, it is imperative that all the interfaces are simple and technical jargon is minimal.
5.2 Back End Architecture

5.2.1 ElementViewer and ElementViewerFactory

These two classes link the Builder tool with JBuilder. Dr. Wenle Zhang, a project team member, authored them.

ElementViewer is basically a NodeViewer - a JBuilder defined class. The NodeViewer interface defines a type of viewer for displaying the contents of Node objects contained in the project pane of JBuilder. When a user opens a node in the JBuilder browser, a tab with the node's display name is added to the top of the browser’s content view. The node may or may not represent a file; the process is the same in either case. At the bottom of the content view is another set of tabs that represent the NodeViewers that have been created by registered NodeViewerFactories that 'accept' the corresponding node.

The ElementViewerFactory is a NodeViewerFactory - a JBuilder defined class. The NodeViewerFactory interface defines a factory that creates instances of NodeViewer for displaying the contents of Node objects.

5.2.2 ElementData

The central class that represents an Element is called ElementData. It contains objects representing all the various components of an Element and performs the parsing of the code for an Element.
The Element is displayed in the structure pane, and following JBuilder’s convention, it is displayed in a tree structure. ElementData extends DefaultTreeModel, and passes itself to a JTree held by the structure pane. The nodes in the tree are pre-defined as “SuperClasses”, “Attributes”, “Buckets”, “Maps”, “Methods”, “Tables”, “Warnings” and “Constants”. These nodes contain leaves, which represent the various components that exist in the Element.

5.2.3 BuilderData

BuilderData was created to be the parent class of all classes that represent Element components like Attributes, Buckets etc. Each component has its own class type and similar components are grouped under common parent classes and ultimately all these are sub classes of BuilderData. AttributeData and BucketData are grouped under a class called FieldData, which holds their common characteristics. AttributeData and BucketData are further sub classed into classes that represent specialized types of Attributes and Buckets. MethodData and MapData are grouped under a class called InnerClassData. MethodData is further sub classed into classes representing the different types of Methods.

In the GUI, the place for these components is in the Element Tree, as leaves. Hence each BuilderData needs to be represented as a TreeNode. Selecting a leaf brings up its details in the Properties Table. The Properties Table is a JTable, which needs aTableModel to populate itself with. Hence each BuilderData needs to be represented as aTableModel as well. The way to fulfill these requirements is to have BuilderData extend
DefaultMutableTreeNode and implement TableModel. Figure 5.1 shows the class hierarchy of BuilderData.
Figure 5.1 BuilderData Hierarchy
5.3 GUI Architecture

The GUI classes account for approximately seventy five percent of the code in the Builder. This is due to the interface-oriented nature of the Builder and also the lengthy nature of GUI coding in Java. The Builder has an abundance of classes that are Dialogs. Some of the other important components of the GUI are the MainPanel, the TreePanel and the Properties Table.

5.3.1 MainPanel

The MainPanel is the central GUI class and an important class in the Builder. It is a JPanel, which contains the embedded Cost Estimator and the Properties Table. It acts as a link between the GUI classes and the back-end classes. All the wizards are contained in this class as objects and other classes can access them from here. It is the MainPanel that creates and holds an instance of ElementData, and is the first class to be instantiated from the Builder. GUI actions like adding of attributes etc. are sent by the GUI components to this class, which in turn passes the command on to ElementData.

5.3.2 Dialogs

There is an abundant use of dialogs in the Builder. Every wizard and many of the editors in the Properties Table are dialogs. All the dialogs have a similar look and feel, right down to the placement of buttons and use of borders.
Most dialogs in the Builder follow a standard mode of operation, which is as follows. When the button “Ok” is hit, the dialog creates an object, which represents the user entered information and hides itself. The other classes then query it and retrieve that object. The next time the dialog is needed, a new instance is not created, and the old instance merely shows itself after resetting and clearing all fields. This makes sure that there are a minimum number of instances floating around and that re-sizing and re-locating of dialogs is preserved. There is only one non-modal dialog, which is the AttributePositionDialog, which displays Attributes being added to a method. This needs to be non-modal because the user needs to select Attributes and drag them to this dialog while it is showing.

A class called the BuilderDialog, extends JDialog, and is the parent class of all dialogs in the Builder. It contains some abstraction of the dialog functionalities. Figure 5.2 shows the hierarchical structure of all the dialog classes in the Builder.
5.3.3 Editing in the Properties Table

The Properties Table is the only place an Element can be edited through the Builder GUI. Considering that it displays around twenty different kinds of BuilderDatas, each of which has an average of eight fields to edit, it needs a very specialized editing class. The class PropertiesTableEditor performs this function. It has one method that returns the appropriate editor and another that returns the edited value directly to the corresponding BuilderData. Each of these methods comprises of highly nested if-else-if structures, which decide what exact type the BuilderData is and accordingly decide what
to return. The value returned to the BuilderData is through a method called “setValueAt()” in each BuilderData.

Even though the Builder has no mechanism to undo changes made through it, no attempt was made to implement this feature simply because JBuilder already provides it. The user can undo any changes made in the Builder by switching to the source tab and hitting JBuilder’s undo.

5.4 Summary of the Primary Functional Structure

Figure 5.3 shows the summary of the basic functioning of the Builder, which can be put in points as follows:

1. The MainPanel creates an instance of the ElementData and stores it as an object.
2. The ElementData is a TreeModel and displays itself within a JTree called the Element Tree
3. The ElementData contains several BuilderData objects within itself. Each BuilderData corresponds to a component of the Element.
4. Each BuilderData object is a TreeNode as well as a TableModel.
5. Each BuilderData is displayed in the Element Tree as a leaf.
6. When a BuilderData is selected using the mouse, its details are displayed in the Properties Table.
7. The details of any BuilderData can be edited using the Properties Table.
8. The MainPanel holds several dialog objects, each of which is a wizard to create a new BuilderData, which will be added to the Element.

9. All changes made to the ElementData are reflected in the source code for the Element and similarly, all changes made to the source code of the Element reflect in the ElementData.

Figure 5.3 Summary of Primary Functional Structure
CHAPTER 6

FURTHER WORK AND CONCLUSIONS

6.1 Further Work

At the time of writing this text the following are some enhancements that could be carried out on the Builder.

1. *Caching unknown Strings*: While parsing the source code, if the Builder finds a string it does not comprehend, it is ignored. Consequently, the corresponding field in the GUI is blank. This could be handled better if the Builder caches the unknown string and displays it as is on the GUI, for the user to interpret and make sense of. Implementing this feature would require a redesign of several of the parsing methods and has hence been left out of the current version.

2. *Detailed Mapping*: The Builder can generate simple relations between Attributes in a Map. It would be desirable to have the Builder create more complex and detailed relations between Attributes.

6.2 Conclusions

The Builder was designed to accomplish the following primary goals.

1. It should cut out the laborious work involved in creating Elements.

2. It should reduce significantly the time taken to create Elements.
3. Code generated through the Builder should be guaranteed to be free of compilation errors.

4. It should provide two-way synchronization between the Element Java code and a graphical representation of the Element.

The Builder has achieved all its primary goals satisfactorily. The Builder generates Java code specific to Cost Elements, following templates set by the Cost Estimator. In this specific task, following are the sub goals that have been achieved by the Builder.

1. Class definition and constructor signatures are generated completely. The user can specify the package for the Element, as well as the classes to be imported into the Element.

2. Twelve different types of Attributes can be created and completely edited.

3. Buckets can be created and edited.

4. BaseMethods can be created and edited. Attributes can be added to and deleted from these Methods.

5. AggregateMethods can be created and edited. Attributes and Sub-Elements can be added to and deleted from these Methods. Sub-Elements can be edited as well.

6. Maps can be created and edited. Simple Attribute associations can be defined.

7. Tables can be created and edited.

8. Warnings can be created and edited.

9. Constants can be created and edited.

10. Html documentation can be generated for all Elements in a package, and links for user added information can be generated in the document.
11. The Element can be dynamically tested during construction, using the Embedded Cost Estimator.

There are some sub goals that have not been achieved, and these are areas that the Builder lacks in. The following lists them.

1. The Builder writes out Attributes, using only the largest constructor signature for that Attribute. So while the user might enter values in only the necessary fields, the Builder defaults the values for all other fields, and writes out the largest constructor. Given that the typical Attribute has around twelve constructor signatures, the task of determining which constructor is the best fit for the user entered information, was considered too arduous, hence the largest constructor is always used.

2. While writing out the Methods, the Builder does not support any information being supplied towards the body of two Java methods that the Methods contain – “calculate()” and “initAttributes()”. “calculate()” defines the algorithm for storing values into the Buckets. These algorithms are specific to each user and as such, it is unfeasible to have an automated mechanism to define this method. “initAttributes()” checks the values of Attributes against the limits set for them, and posts Warnings if any conflicts are discovered. This feature was low priority and hence did not get implemented in this version of the Builder.

3. Maps can define only direct Attribute associations. In several Elements, Attribute associations are not direct and there could be a complex function mapping the
value of one Attribute to the value of another. The Builder does not handle this case.

4. Tables are often initialized in static blocks in the code. The Builder can write such Tables, but does not have the ability to parse them. This shortcoming is due to JOT not supporting the parsing of static blocks. If in a future release of JBuilder, this problem is tackled, the Builder will then be in a position to parse Tables that are initialized in static blocks.

In conclusion, while the Builder doesn’t eliminate the need for manually entered code, it definitely reduces the time and effort required to build an Element, to an appreciable degree.
REFERENCES

<http://java.sun.com/j2se/1.3/docs/api/>.


[4] Dr. R.P. Judd, Dr. David Koonce, Dr. Dusan Sormaz, “The Cost Estimator Documentation”.

[5] Dr. R.P. Judd, Dr. David Koonce, Dr. Dusan Sormaz, “Cost Estimator Training (Introduction) Power Point presentation”.


<http://www.ist.co.uk/visaj/index.html>.


http://www.xintegra.com/content/products/product_list.html.


This section will detail the steps to be followed in order to create the Java code for an Element using the Builder. The purpose of this section is to make clear the functionality of the Builder.

The Element that will be used for the purpose of illustration is an “Arm” for a Chair. The Arm will be wooden, consisting of an armrest and two supports to hold it up. Elements are usually developed as part of a family. In the construction of this Arm, it will be assumed that the family of Elements for the Chair is well developed. The Arm will be an addition to this family. Two pre-existing Elements from this family that will be required for the construction of this Arm are:

1. SystemElement: Elements such as these are usually the first to be created in an Element family. They contain a level of abstraction of the components that are likely to be used by several Elements in the family. SystemElement contains some Attributes like “material”, some Buckets and Tables among other things. It has been designed such that all Elements in this family can extend it.

2. WoodPiece: This Element represents a simple, plank-like wood-piece, and has Methods defined to calculate its cost depending on its Attribute values. This is designed to be a low level Sub Element that makes up various Elements in the Chair family like the Back, Front etc. The idea is to break the Element into
simpler Sub Elements, which are easier to apply costing algorithms to. This philosophy will be clearer after this exercise.

A.1 Steps to Building an Element

The following are the steps to build a new Element using a simplistic approach. The process of building an Element is typically far more complex than this. This approach has been kept simple for the purpose of illustration.

1. **Identify Attributes to describe the new Element.**

   Figure A.1 shows the Arm this exercise intends to create and the Attributes identified for it.

   ![Figure A.1 The Arm and its Attributes](image)

   Four attributes have been identified for this Arm – Length, Height, Width and Material.
2. **Identify Sub-Elements of the Element**

One of the Methods used for estimating cost, is the Aggregate Method. This method breaks up an Element into its constituent Sub-Elements, and then relies on the cost estimation of each of the Sub-Elements to accrue into the cost of the Element itself. Figure A.2 shows the Sub-Elements the Arm is constituted of.

![Figure A.2 Sub-Elements of the Arm](image)

The Arm consists of three Sub-Elements that are grouped into two types. The first is the “ArmRest” which is a WoodPiece. It has been identified to have four Attributes – Length, Width, Depth and Material. Second is the ArmSupport,
which are two in number. The ArmSupport is a WoodPiece as well and has four Attributes similar to those of ArmRest.

3. **Determine Mappings between Element Attributes and Sub-Element Attributes.**

   When a relation exists between the values of certain Element Attributes, and the values of certain Sub-Element Attributes, they are defined using Maps. From inspection of Figure A.2, the following relations can be arrived at.

   A. Between the Arm and ArmRest:

   \[
   \text{ArmRest.Length} = \text{Arm.Length} \\
   \text{ArmRest.Width} = \text{Arm.Width} \\
   \text{ArmRest.Material} = \text{Arm.Material}
   \]

   B. Between The Arm and ArmSupport

   \[
   \text{ArmSupport.Width} = \text{Arm.Width} \\
   \text{ArmSupport.Length} = \text{Arm.Height} - \text{ArmSupport.Depth} \\
   \text{ArmSupport.Material} = \text{Arm.Material}
   \]

   Mappings are not always direct relations as in the above examples. They can be complex functions relating one Attribute to another.
4. **Determine the Buckets to store the cost.**

   It needs to be decided, what calculated outputs for cost, our methods will produce. In this example, all the Buckets needed are defined in the SystemElement; hence there is no need to declare any Buckets. This illustration will use an AggregateMethod for cost estimation. This means the costing will be done at the SubElement level, or WoodPiece level for this example. The WoodPiece class contains predefined costing algorithms, which need Attribute values to work. These Attribute values are obtained from the Maps relating the Arm to the WoodPiece SubElements. The AggregateMethod at the Arm level will aggregate the buckets at the WoodPiece SubElement level and hence calculate values for the Buckets corresponding to the Arm.

With the four steps needed to build this Element now defined, what follows is a look at how these would be executed using the Builder.

**A.2 Creating the Arm Element**

The following is a step-wise description of what it would take to create the Arm as described above.

1. **Create the Shell for the Arm, using the New Element Wizard.**
Bring up JBuilder and click on the “New” button on its tool bar or go through the menu, selecting “File” and then “New”. This will bring up the dialog shown in figure A.3

![Object Gallery](image)

**Figure A.3** JBuilder Dialog to select the type of the New Object

Select “Cost Element” from the icons and click “OK”. This will bring up the New Element Wizard (Figure 4.2). Make entries into the fields of the wizard as shown in figure A.4, and click “OK”
The package name depends on the user entirely. The above wizard will produce a Java file for the Arm and create a shell of code for it. Clicking on the CE Builder tab will fire up the Builder, which can edit the Arm. Figure A.5 shows how the Builder renders the Arm at this point.
Figure A.5 Shell of the Arm rendered in the Builder

The Element Tree displays several Buckets that have been inherited from SystemElement along with the Attribute “Material” and other Attributes. The other three Attributes for the Arm need to be defined.
2. Define the Attributes for the Arm

Click on the “New Attribute” button on the Builder ToolBar to bring up the New Attribute Wizard (Figure 4.4). Make entries into the wizard as shown in the figures A.6 through A.8, one for each Attribute to be created.

Figure A.6 New Attribute Wizard declaring Length for Arm
Figure A.7 New Attribute Wizard declaring Width for Arm

Figure A.8 New Attribute Wizard declaring Height for Arm
These three operations will result in three Attributes being created and added to the Arm.

3. **Define the Sub-Elements for the Arm**

Sub-Elements for an Element are defined within a Method, more specifically an Aggregate Method. Click on the “New Method” button on the Builder ToolBar to bring up the New Method Wizard (Figure 4.6 and Figure 4.7). Make entries on this wizard as shown in figures A.9 and A.10.

![Figure A.9 New Method Wizard (page 1) declaring a Method for the Arm](image-url)
This operation will define a method called “Detailed”. This wizard does not have the ability to add Sub-Elements to the Method. This must be done so through the Properties Table. Figure A.11 shows the Builder rendering the Arm at this point. The “Detailed” Method is selected in the Element Tree and its properties are seen in the Properties Table. Clicking on the row that says “Added Sub-Elements” will bring up a specialized dialog to handle the Sub-Elements associated with the Arm through this Method.
Figure A.11 The Builder rendering the Arm and showing the properties of the selected Aggregate Method

Clicking on the value corresponding to the “Added Sub Elements” row, in the PropertiesTable, brings up the dialog shown in figure A.12.
Since there are no Sub-Elements at this point, the dialog has none listed. Click on the “Create New” button to bring up the dialog to create a new Sub-Element, shown in figures A.13 and A.14. The first figure shows the dialog creating an ArmRest and the second an ArmSupport. The class for both has been selected as WoodPiece, and the Maps for both have been left as “null” for the time being. These two operations will create the two types of Sub-Elements the Arm needs.
Figure A.13 New Sub-Element Wizard creating an ArmRest

Figure A.14 New Sub-Element Wizard creating an ArmSupport
At this point the Sub-Elements have been added to the Method and the Added Sub-Elements dialog will now look appear as shown in figure A.15.

![Added Sub Elements Dialog](image)

**Figure A.15 The Added Sub Elements Dialog after adding Sub-Elements**

4. **Define the Mapping between the Arm and its Sub-Elements.**

   With the Sub-Elements being added, the mappings between the Attributes of the Arm and the Attributes of the Sub-Elements need to be defined. Clicking on a cell in the column “Map”, in the Added Sub Elements Dialog, will bring up a dialog to set the Map for the Sub-Element corresponding to that row. Click on the cell in the row corresponding to ArmRest and in the column “Map”, to bring up the dialog shown in figure A.16.
A pre-existing Map can be selected or a new one created. Click on “Create New” button to bring up the New Map Wizard (Figure 4.8). Make entries in this wizard as shown in the figure A.17, to create a Map for the ArmRest. Hit the “Attribute Associations” button on this wizard to bring up the dialog shown by figure A.18. This dialog lists all the Attributes of the Element and of the Sub-Element, and allows the user to make direct associations between them. Use this dialog to define the following associations:

\[
\text{ArmRest.Length} = \text{Arm.Length} \\
\text{ArmRest.Width} = \text{Arm.Width} \\
\text{ArmRest.Material} = \text{Arm.Material}
\]
Figure A.17 New Map Wizard creating Map for ArmRest

Figure A.18 Dialog to Define Attribute Associations from the Arm to ArmRest
The same procedure needs to be followed to create a Map for the ArmSupport.

The Associations in this case are:

\[
\text{ArmSupport.Width} = \text{Arm.Width}
\]

\[
\text{ArmSupport.Length} = \text{Arm.Height} - \text{ArmSupport.Depth}
\]

\[
\text{ArmSupport.Material} = \text{Arm.Material}
\]

Make entries into the New Map and Attribute Association wizards as shown in figures A.19 and A.20.

Figure A.19 New Map Wizard creating Map for ArmSupport
Figure A.20 Dialog to Define Attribute Associations from the Arm to ArmSupport

The association between ArmSupport.Length and Arm.Height has not been defined completely by the dialog in figure A.20. The subtraction of the value of ArmSupport.Depth will have to be coded manually.

Since there are two ArmSupports, the quantity of the Sub-Element-ArmSupport, needs to be declared as two. This is done by manually coding this information into the source code. At this point 74 lines of Java code representing the Arm have been written out, of which only 2 had to be manually entered.
A.3 Testing the Arm Element by using the Embedded Cost Estimator

The embedded Cost Estimator is executed to test the Arm that has been constructed. Figure A.21 shows the Builder rendering the completed Arm, with the Embedded Cost Estimator displaying the estimated cost for it.

The estimate of total cost (listed under the Bucket “burden”) is displayed in the Navigation Panel, with a break-up of the cost between the Sub-Elements. The Summary Panel shows a more detailed synopsis of the Buckets, and the Pie Chart represents graphically the distribution of the total cost between the Sub-Elements. The Method Panel lists some of the Attributes, as chosen during the construction of the Detailed Method. Changing the values on any of these Attributes, dynamically updates the estimate.

In conclusion, an Element has been constructed using the Builder, and has been dynamically tested by the Embedded Cost Estimator.
Figure A.21 Builder Rendering the completed Arm, while executing the Embedded Cost Estimator