5-1-1998

Exploring design patterns with the Java programming language

Stephanie Burton

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
RELEASE PERMISSION FORM

Rochester Institute of Technology

Implementation of Design Patterns in the Java Programming Language

I, Stephanie Burton, hereby grant permission to any individual or organization to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only.

Stephanie P. Burton

May 19, 1998

Date
ABSTRACT

This project describes and discusses the concepts of design patterns giving a historical background as well as citing contributions to the software development field and current research efforts. In addition, the advantages and disadvantages of using design patterns and efforts to encourage design pattern usage in software organizations are discussed.

Seven design patterns (Builder, Adapter, Composite, Template Method, Façade, Mediator and Strategy) were chosen as the basis for example concrete applications. The concrete applications are given in the form of design documentation, source code and executable software. The applications demonstrate the use of design patterns in developing object oriented software applications. The applications are implemented in the Java programming language.

The Java programming language was used because it is a popular object oriented programming language. An aspect of its popularity comes from its ability to execute with Java enabled browsers on a variety of computing platforms. Noted researchers in the area of design patterns assert that design patterns are language independent, however, much of the implementation in the area of design patterns has been written in the C++ language. The contribution of this project lies in implementing selected design patterns in Java and noting experiences that support or refute the conjecture that design patterns are language independent.
# Table of Contents

Table of Contents .............................................................................................................. ii
List of figures ....................................................................................................................... iv
Glossary ............................................................................................................................... v
Notation Definition ............................................................................................................. vii

1 Introduction ......................................................................................................................... 1

2 Background and Concepts ............................................................................................... 4
   2.1 Software Development ............................................................................................... 4
   2.2 Christopher Alexander ............................................................................................. 6
   2.3 Patterns in Software ................................................................................................ 7

3 Contributions to the Software Community .................................................................... 12
   3.1 Object Interaction ..................................................................................................... 12
   3.2 Abstraction .............................................................................................................. 14
   3.3 Vocabulary .............................................................................................................. 15
   3.4 Design Expertise .................................................................................................... 16
   3.5 Repositories .......................................................................................................... 17
   3.6 Process/Organization ............................................................................................. 19

4 Advantages and Disadvantages ...................................................................................... 21
   4.1 Software System Description ................................................................................. 21
   4.2 Software Design ...................................................................................................... 22
   4.3 Design Pattern Novelty ......................................................................................... 23
   4.4 Training .................................................................................................................. 23
   4.5 Software Development Processes .......................................................................... 24

5 Acceptance and Adoption ............................................................................................... 26
   5.1 Training Efforts ....................................................................................................... 26
   5.2 Pattern Mining ....................................................................................................... 28
   5.3 Pattern Advocacy ................................................................................................. 30

6 Pattern community Activities ......................................................................................... 32
   6.1 History .................................................................................................................... 32
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Class Diagram Notation</td>
<td>vii</td>
</tr>
<tr>
<td>1</td>
<td>Structure Class Diagram for Builder Pattern</td>
<td>39</td>
</tr>
<tr>
<td>2</td>
<td>Class Diagram for HTML Text Converter</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>Structure Class Diagram for Composite Pattern</td>
<td>46</td>
</tr>
<tr>
<td>4</td>
<td>Class Diagram for Dependency Traverser</td>
<td>47</td>
</tr>
<tr>
<td>5</td>
<td>Example of “Makefile.txt”</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>Structure Class Diagram of the Adapter Pattern</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>Class Diagram for Image Button Adapter</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>Structure Diagram for Mediator Pattern</td>
<td>58</td>
</tr>
<tr>
<td>9</td>
<td>Class Diagram for Font Displayer Applet</td>
<td>59</td>
</tr>
<tr>
<td>10</td>
<td>Mediator App GUI Window</td>
<td>61</td>
</tr>
<tr>
<td>11</td>
<td>Structure Class Diagram for Strategy Pattern</td>
<td>64</td>
</tr>
<tr>
<td>12</td>
<td>Class Diagram for Graph Manipulator</td>
<td>65</td>
</tr>
<tr>
<td>13</td>
<td>Structure Class Diagram for Template Method Pattern</td>
<td>70</td>
</tr>
<tr>
<td>14</td>
<td>Class Diagram for Shortest Pathfinder</td>
<td>71</td>
</tr>
<tr>
<td>15</td>
<td>Structure Class Diagram for Façade Pattern</td>
<td>75</td>
</tr>
<tr>
<td>16</td>
<td>Class Diagram for Graph Request Handler</td>
<td>77</td>
</tr>
<tr>
<td>17</td>
<td>Graph Request Handler GUI Window</td>
<td>79</td>
</tr>
</tbody>
</table>
GLOSSARY

Abstract class, page vii
“A class whose primary purpose is to define an interface. An abstract class defers some or all of its implementation to subclasses. An abstract class cannot be instantiated” [4, p. 359].

Aggregate object, page 46
“An object composed of subobjects. The subobjects are called aggregate’s parts, and the aggregate is responsible for them” [4, p.359].

Black-Box, page 8
“Composed objects reveal no internal details to each other” [4, p. 359].

Class, page 1
“A class defines an object’s interface an implementation, their internal structure and operations and the static relationships between” [4, p. 359].

Concrete class, page vii
“A class having no abstract operations. It can be instantiated” [4, p. 359].

Design pattern, page 2
“A design pattern systematically name, motivates, and explains a general design that addresses a recurring design problem in object oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context” [4, p. 360].

EuroPLoP, page 34
European Pattern Languages of Programming, an annual conference during which issues concerning patterns and pattern language development are presented and discussed.

Gang of Four (GOF), page 33
The name given to the four authors of the book, Design Patterns: Elements of Reusable Object-Oriented Software

GUI, page 59
Graphical User Interface; A display method allowing user to interact with an application using visual devices such as windows, buttons, checkboxes, sliders, etc.

Hillside Group, page 33
Group concerned with the use of software design pattern in the software community.
**HTML**, page 29
Hyptertext Markup Language; A textual format used to describe pages displayed on the World Wide Web.

**Inheritance**, page 17
“A relationship that defines one entity of another. Class inheritance defines a new class in terms of one or more parent class. The new class inherits it interface and implementation from its parents. The new class is called a subclass or derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or existing implementations” [4, p. 360].

**Interface**, page viii
“The set of all signatures defined by an object’s operations. The interface describes the set of requests to which and object can respond” [4, p. 361].

**Object**, page viii
“A run-time entity that packages both data and the procedures that operate on that data” [4, p. 361].

**OMT**, page vi
Object Modeling Technique. A notational convention used to represent object-oriented classes, objects and their relationships.

**OOPSLA**, page 32
Object Oriented Program Systems and Applications, an annual conference during which issues of software engineering and software development are discussed and presented.

**Override**, page 39
“Redefining an operation (inherited from its parent class) in a subclass”[4, p. 361].

**PLoP**, page 34
Pattern Languages of Programming (PLoP), an annual conference during which issues concerning patterns and pattern language development are presented and discussed.

**Subclass**, page viii
“A class inherits from another group of classes that collaborate to fulfill a set of responsibilities” [4, p. 361].

**Subsystem**, page 8
“An independent group of classes that collaborate to fulfill a set of responsibilities”[4, p. 361].

**Type**, page viii
“The name of the particular interface”[4, p 362].
The notation used in by Erich et al [4] serves as the standard notation for this paper. Figure 0.1 displays the notation conventions used in the class diagrams appearing in this paper.

Figure 0.1 - Class Diagram Notation [4, p. 365]

Figure 0.1a shows the notation for the abstract and concrete classes. According to OMT (Object Modeling Technique) notation, a box with a class name at the top denotes a class. The class operations are listed below the class name. The abstract class has a triangle in the upper right-hand corner (a deviation from Gamma's notation).
"In some design patterns it's helpful to see where client classes reference Participant classes. When a pattern includes a Client class as one of its participants (meaning the client has some responsibility in the pattern), the Client appears as an ordinary class. When the pattern does not include a client participant (i.e., client has no responsibility in the pattern), but including it nevertheless clarifies which pattern participants interact with clients, then the Client class is shown in gray" [4, p. 364], as shown in Figure 0.1 b.

"Figure 0.1c, shows various relationships between classes. The OMT notation for class inheritance is a triangle connecting a subclass (LineShape in the figure) to its parent class (Shape). An object reference representing a part-of or aggregation relationship it indicated with by an arrowheaded line with a diamond at the base. The arrow points to the class that is aggregated (e.g., Shape). An arrowheaded line without the diamond denotes acquaintance (e.g., LineShape keeps a reference to a Color object). A name for the reference may appear near the base to distinguish it from other references" [4, p. 364].

The "creates" relationship shows which classes instantiate other classes. A dashed arrowheaded line denotes this relationship. The arrow points to the class that is instantiated. In Figure 0.1c, the CreationTool creates LineShape objects [4, p. 364].

"OMT defines a filled circle to mean 'more than one'. When the circle appear at the head of a reference, it means multiple objects are being referenced or aggregated. Figure 0.1c shows that Drawing aggregates multiple objects of type Shape" [4, p.364].

Pseudocode annotation let the designer sketch the outline of a method's implementation. Figure 0.1d shows the pseudocode annotation for the Draw operation on the Drawing class.
INTRODUCTION

Software has provided solutions to a variety of problems in many industries. During the discovery of those solutions, software developers have learned a great deal about a variety of problem domains and application requirements. Developers use knowledge gained from previous projects when exploring solutions to new problems. Recognition and application of existing solutions is beneficial to software developers.

Throughout the history of software development, developers have endeavored to capture the essence of a problem solution. However, the means of capturing needed information was limited to the basic components of the chosen methodology. The structured programming methodology, represented in languages such as Fortran and C, divided the problems into a hierarchy of functional tasks that worked together to produce the desired output. Developers, using structured programming techniques, captured their solutions through the creation of useful functions that were essential to the problem solution and as compact as possible. The Object-Oriented (OO) programming methodology, represented by languages such as Eiffel, C++, and Pascal, group data and functions into modular units which exhibit a behavior that is essential to the solution and distinct. OO developers capture the essential parts of the solution in well-defined classes and class hierarchies.

Regardless of the chosen methodology, all developers do not encounter the same problems or have knowledge of the same set of existing solutions. For example, new software developers lack the experience gained through encountering a variety of problems
and solutions. So many times, they rediscover existing solutions through the application of basic principles. The rediscovery is often difficult and time consuming. A method to capture problem solutions is still needed in the software development community. Research in the area of design patterns is exploring a useful means of capturing software solutions to reoccurring problems. Design patterns present the essential components of a problem solution as well as the relationship between the components and the constraints of the solution. The solutions are presented in general terms leaving the concrete implementation details to the implementor. Design patterns research has the potential to provide valuable contributions to the software community.

This project describes and discusses the concepts of design patterns giving a historical background as well as citing research efforts and contributions to software development field. Examples of concrete applications are given in the form of design documentation, source code and executable software. The applications are implemented in the Java programming language. The applications demonstrate the use of design patterns in developing object oriented software applications.

The concept of design patterns has existed for over a decade; however, it has been slow to gain widespread use until recently. Much of the widespread use has been among experienced software engineers. This project presents example applications using design patterns as a tutorial for inexperienced software developers. The sample applications encourage new programmers to take advantage of existing expertise in design patterns early in their careers and expose new programmers to the Java programming language.

The Java programming language was used because it is a popular object oriented programming language. An aspect of its popularity comes from its ability to execute with
Java enabled browsers on a variety of computing platforms. Noted researchers in the area of design patterns assert that design patterns are language independent, however, much of the implementation in the area of design patterns has been written in the C++ language. The contribution of this project lies in implementing selected design patterns in Java and noting experiences that support or refute that design patterns are language independent.
2 BACKGROUND AND CONCEPTS

2.1 Software Development

A software system is a product of a development process. Many models exist that attempt to capture the essence of the software development process. One of the most widely known models is the Waterfall model. The Waterfall model of software development identifies the major phases of software developments as requirements gathering, analysis, design, implementation, and maintenance. The stages are depicted as sequential steps in a process beginning with requirements and ending in the maintenance phase. Each phase has a defined beginning and end. In addition, the next phase does not begin until the previous phase has been completed. Other models exist, such as the Spiral model that follows similar views with additions. The Spiral model suggests that development proceed in iterations of the Waterfall model. A completed iteration produces a working version of the system containing a subset of the complete feature set. Remaining features are added to the system during subsequent iterations until the system is complete.

These models aid all levels of a software project. At the highest levels, these models help executives determine the end date of the project. Managers use the models in assessing resources. At any given point in the project, developers are aware of the tasks that they should be performing. This highly organized description of software development models only scratches the surface of today’s software development practices.

According to Gabriel, “...the truth is that today the production of software is barely controlled chaos...”[1, p 1] Chaos evolves from instabilities in any stage of the
software development process. In large-scale systems, there are numerous and sometimes conflicting requirements that must be designed into a single system. Also, ad-hoc design practices utilized by variably skilled developers may lead to deficiencies in different areas of the system. These instabilities arising in the design phase, for example, may also negatively affect other phases, such as implementation and maintenance.

Chaos in the software development process affects the final software product. Historically, large-scale software systems have been delivered behind schedule and beyond budgetary plans. The chaotic development process leads to erroneous planning and resource allocations. In addition, the quality in some system has been less than what was promised at the beginning of the project. Systems have been delivered with numerous defects. Multiple releases are expected before the system is deemed reasonably stabled.

Developers are key assets in the production of software. However, there exists a gap between academic training and industry practice. Students develop most of their skills on small-scale projects and isolated programming examples. Through these examples only limited experience is gained. The study of large-scale systems is also limited. Gabriel asserts that "In schools, there is no such thing as studying successful designs and implementations of large systems"[1]. He goes on to suggest that even among experienced developers, there is no agreement of a "good" design or implementation. Thus, novice developers gain most of their experience through practice rather than training. The accumulation of experience through practice takes time and a large number and variety of projects.

In light of the difficulties, software has become an accepted risk in many industries. The potential gains somewhat offset the difficulties. Many software systems last well beyond their initial development. The flexibility of software allows it to change with
its environment to incorporate new technological advances and changes in societal needs and requirements. The returns of software have increased the demand for it.

As industries increase the use of software, many researchers have endeavored to discover ways to introduce stability into the development process to insure its longevity. Some researchers have concentrated on the design phase of the software development process. In such a short time, software's popularity has grown immensely resulting from its use in a variety of industries. Thus, some researchers have endeavored to capture the design knowledge gained thus far in the software community. With the reuse of tried and tested designs, other phases of the development process may be positively affected. In addition, new developers can be exposed to the existing expertise early in their careers, which positively affects the development process.

2.2 Christopher Alexander

In the search for stable practices in design, some researchers have looked to other long-standing engineering disciplines as a guide. Researchers seeking to improve the design phase of software development have focused on the works of Christopher Alexander.

“Christopher Alexander is an architect, builder, and professor of architecture at the University of California, Berkley” [1, p. 5]. He has written a three-book series that presents theories of architecture, building and planning. The titles in the series are The Timeless Way of Building, A Pattern Language, and The Oregon Experiment (all from Oxford University Press).
Alexander used patterns to describe structures that were visually pleasing and provided functionality characteristics for their intended use [6, p. 20]. The patterns that he developed describe solutions to problems reoccurring in design in terms of the forces involved in solving the problem. Patterns aided in the design of each project segment. Alexandrian patterns did not reveal exactly how to construct the structure. This enables creativity within the bounds of defined constraints. Alexander compiled a collection of patterns into a pattern language.

“Alexander has formulated what he calls a pattern language for building and cities” [3, p. 1]. Each entry in the pattern language is a pattern that describes a solution to a reoccurring design problem. Alexander ordered his patterns in such a way that a finite number of options are presented at each step in the design project. An Alexandrian pattern contains the following information: the definition of the problem, definition of the problem context, description of the forces involved, description of the solution, resolution of the forces and a design rationale.

2.3 Patterns in Software

“A little imagination reveals rich analogies between these architectural patterns and the structure of programs” [2, p. 20]. Buildings evolve in piecemeal fashion as the needs of the inhabitance change over time [1, p. 2]. Programs evolve and change over time as new technologies are discovered or new requirements are introduced. Requirement changes are incorporated into the existing structure of the program introducing changes to specified areas of the program just as additional rooms are added to the structure an existing building. Both areas are concerned with the “access between the components” [2,
The software community is interested in the architectural development of patterns because both areas have similar structural concerns and like architecture, some problems reoccur in many projects.

Members of the software community have adopted the notion of patterns and tailored patterns to meet the needs of software. One of the most noted efforts to gather a set of patterns for software began as a part of Erich Gamma's Ph.D. thesis. Following the introduction, three additional people – Richard Helm, Ralph Johnson, and John Vlissides – joined Gamma to continue development of a catalogue of software patterns. To distinguish software patterns from Alexander's architectural patterns, the software patterns were given the name design patterns. Gamma et al defined a design pattern as: “A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object oriented systems. It describes the problem, the solution, when to apply the solution and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context” [4, p. 360]. The qualifying term “design” emphasizes “the capturing of design expertise as opposed to other software development skills such as domain analysis or implementation” [5, p.37].

The definition and presentation of design patterns by Gamma et al [5] has far reaching implications in software. Gamma mentions that the solutions are applicable within a particular context. Software has different levels of views or granularity. One can view software on a system level where a system is treated as a black-box and only the outward behavior is of importance. One can view software on a subsystem modular level
where cooperation subsystems provide functionality within a large system. Software can be viewed on the level of its internal behavior or code level implementation. Each of these views is a distinct context where solutions may be applied. Since the design pattern defines its context, design patterns can describe solutions for different granularities of software. Patterns can describe solutions applicable to a complete system, a component within a system, or an implementation detail, like iterating through a list.

The representation of a pattern in a literary document is very useful. The text provides a wealth of information to which graphical representations complement. Neglecting implementation dependencies when describing the elements of a solution enables the implementation of the solution in any object oriented language. In addition, the solutions may also be applied in non-object oriented systems because the solution of the problem lies within the components and their relationships to each other rather than the structures, such as class diagrams, that they are presented.

Each pattern is presented in a prose document. Gamma describes each pattern in a thirteen-section format containing both text and diagrams. The format is structured as follows: Name, Intent, Also Know As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns. The thirteen sections can be grouped based on the information that they provide. “The first three sections identify the pattern. Section four approximates the contents of an Alexandrian pattern: It gives a concrete example that illustrates the problem, its context, and its solution. Sections five through nine define the pattern abstractly.” Sections ten and eleven give a concrete example of the implementation of the solution. Sections twelve and thirteen provide references to the pattern [5, p. 37].
As the software community has tailored the notion of patterns to the needs of software, the application of patterns is also slightly different than the Alexandrian application. Alexandrian patterns are applied in sequence; however, the Gamma patterns are presented in no particular order. There is not direction to the “right” pattern. In contrast to Alexander’s brief pattern descriptions, Gamma patterns include the Alexandrian pattern characteristics and includes additional abstract descriptions and implementation examples. The chosen format enables “quick reference in the heat of design or implementation” [5, p. 37]. The implementation presented can be followed as a recipe for similar implementation projects. However, it should be noted that the implementation presented in the pattern description is only an example of one possible implementation and should not be confused as the sole representation of the pattern.

Design patterns are useful tools for developers at any stage of their practicing careers. For novices, design patterns presents expert experience gathered by other practitioners over time and has been tested through use in numerous applications. They encourage creativity and exploration with defined constraints. “Design patterns build them [novices] up to the experts way of thinking” [2, p. 66]. For experienced designers, design patterns serve as a reminder of the current knowledge that already exists. Design patterns help these designers concentrate on other aspects of design rather than spending time rediscovering existing knowledge. For domain experts, design patterns represent a body of existing knowledge to which the expert may augment. Domain experts have the experience and knowledge to recognize deficiencies and omissions within the existing knowledge base. To the software community at large, design patterns represent the accumulated work and experience of some practitioners that can serve as a repository of
information “in a set format under a standard nomenclature” [3, p. 1].

In summary: Although the Waterfall and Spiral models exist to theoretically describe the process of software development, the software development process proceeds along an apparently chaotic path. The resulting software systems are delivered late and over budget due to the lack of sufficient planning for such a chaotic and unpredictable process. The delivered systems have low quality due to existing errors. Novice developers are introduced to the process with limited training and experience.

In an effort to harness existing knowledge about various software systems, researchers have looked to architecture and adopted Alexander Christopher's notion of patterns that describe the solution to recurring problems. In software, design patterns can introduce some stability when applied during the design phase of the software development process by outlining or injecting stable solutions into new or existing projects. They may also positively impact the other phases of the software development process by representing a repository of known solutions for reuse, providing guidance during implementation and maintenance phase of the development process, and positively affecting the developers participating in the process. Design patterns are useful tools at any stage of a designer's career.

Since parallels exist between architecture and software, adopting some of the concepts is not such as far-fetched notion. Software design patterns describe recurring software design problems in a prose document that contains the characteristics of Alexandrian patterns augmented by concrete implementation details and examples. Software design patterns lack the sequential application of the Alexandrian patterns but the added information provides for easy reference during design and implementation.
Contributions to the Software Community

The concept of design patterns was slow to be accepted by the software community. At first, only small groups within the software community reaped the benefits of the design pattern concepts. As discussion and practical use increased, many realized that design patterns could make valuable contribution to the software community. Six such contributions that design patterns offer the software community are discussed.

3.1 Object Interaction

The object methodology focused on developing good classes and class hierarchies to capture behavior. Coplien asserts that “adopters of the object paradigm focused on the ‘find-the-object’ exercises, deferring or forever losing the system perspective of interactions between objects”[6, p. 36]. Objects (or classes and their hierarchies) are almost never used in isolation. Also, it is very rare that a single object would contain the behavior of an entire system. Software systems encompassing multiple concerns are best addressed with more than one object. However, the participants in the system are not the only important factors when expressing the behavior of a system. Interactions between the objects in the system distinguish a system from a mere collection of isolated objects.

Design patterns encapsulate the protocol of interaction between objects that could not be expressed by a single object [7, p. 48]. Individual design patterns capture the solutions to recurring problems in software design. In this case, the solution to a specific problem specifies the system. On a small scale, the components of in the solution may be
objects. Interaction between the objects may be creating one another or passing messages to one another. On a larger scale, the components of the solution may be distinct subsystems interacting together. "Such composition provides and abstract view of a system, so that a designer can do system-level analysis and reason about system integrity constraints" [7, p. 46]. At this level, interactions between the components or sub-systesms form some global foundations of the overall operations of the system being designed.

Individual patterns can be organized into groups called pattern languages. Pattern languages represent a collection of solutions to reoccurring design problems within a domain. According to Alexander pattern languages provided a guide for the application of individual patterns. Pattern languages demonstrate the interactions (in the form of dependencies) between individual patterns as they relate to constructing a system. This represents yet another scale upon which patterns can convey interaction between objects in a system.

Although identifying the components of a system is important, the interaction between the components of a system contributes greatly to describing the behavior of a system. In the collection of patterns presented by Gamma, some of the patterns have a similar structure of components in the solution of the problem. However, clear distinction between the purpose and intent of the patterns can be made once the interactions between the components are understood. From individual patterns to pattern languages, patterns describe interactions between components regardless of how the integration scale changes the focus of the components within the system under examination.
3.2 Abstraction

The expression of abstraction is one of the founding principles of the object methodology. Abstraction allows developers to defer the considerations of the detailed implementation until late in the design stage by representing the components in the system as interfaces. In some cases, this enables adequate management of complexity by minimizing the overload of details. Design Patterns lends support to the concept of abstraction.

Design patterns describe a general way of solving a problem. The documentation of the pattern can be viewed as the description of an interface of the solution. The interface of the solution is presented in the form of text and diagrams. Together these approaches convey the structure of the solution's interface, in the form of a general view of what each component's interface should contain, and the behavior of the solution, in the form of explanations and examples. Design patterns give a thorough explanation of the solution's interface while leaving the developer to solidify the solution. The developer is left to choose the implementation language and manage the implementation details.
For systems that incorporate multiple solutions, design patterns help the developer to organize solutions in a way to achieve a system with some desired behavior. Design patterns can represent a modular abstraction, which hides the internal objects. Dealing with larger modular abstractions enables developers to concentrate on organizing the solutions into a sensible system. The modular view lets the developer temporarily circumvent the total number of actual objects that must be present in the system. Temporary deferral of some smaller details may result in a system better organized and easier to maintain and extend because the system design was not driven by its implementation.

3.3 Vocabulary

The software community is a fairly dynamic community. The community evolves and changes as new discoveries and innovations are made. Over time, new buzzwords become popular while traditional terms take on new meanings. The constant change in terminology can become confusing even to the practitioners who are faced with the implications of the terminology on daily basis. Design Patterns introduce a consistent vocabulary among its users.

Design patterns introduce and define vocabulary in a variety of ways. Like Alexander, patterns produced by Gamma and others have a name. The name serves as an identifier for the concepts described in the pattern. In the descriptions of the concepts in the pattern other terms are introduced. These terms may be specific to the pattern or signify a common link between one or more design patterns. In any case, the words used to describe the pattern become a part of a user’s vocabulary.
Introduction of a consistent vocabulary facilitates easier communication between users of the same set of patterns. The mere mention of a pattern name may immediately conjure up the underlying concepts of the pattern. Since both parties have accepted the same vocabulary base, the unspoken concepts that each envisions is consistent with one another. Over time with continued use, the pattern language vocabulary becomes second nature to the users. The vocabulary becomes a form of abstraction. Patterns give rise to a vocabulary that enables developers "to talk about structures larger than modules, procedure, or objects"[6, p. 37]. The users can concentrate on issues of a higher level while maintaining ties to the underlying concepts. Working toward a more effective means of communication between users helps to more clearly convey concepts between team members while minimizing misunderstandings due to differing foundations and assumptions.

3.4 Design Expertise

Software is used in numerous products in a variety of industries. Through the development of such a wide variety of product, valuable knowledge and expertise has been gained in the software community. A popular way of transferring knowledge is through examining source code. Design patterns capture the subtleties of software design solutions that may not be apparent from examining source code.

The design pattern documentation presents the solution to a design problem in diagrams and textual explanations. The textual explanations reveal, for example, the problem motivating the solution. It may also present variations of the problem for which the pattern may also apply. The concise structural and behavioral diagrams depict the
relationships and interactions between the components of the solution. Together, this information provides a strong foundation for understanding and incorporating the solution into one's own work. This documentation also provides a clear direction that is useful for initial development and can serve as a high level overview for maintainers. Implementation issues pertaining to the application of the pattern is presented as consequences. The consequences represent discoveries made through the various applications of the pattern. The consequences present trade-offs and limitations that set the boundaries for the solution. This enables developers to make informed choices as to whether the solution meets the needs of the applications now and in the applications future. In addition to providing information to make good design choices, design patterns aide in building confidence. Design patterns represent information gathered during several applications. Thus, the user can feel some degree of sureness in justifying its use. In addition developing confidence among developers encourages them to proceed to the next level of practice where they may discover new innovations.

Design expertise is a step beyond merely knowing the concepts of specific subject matter. It entails certain knowledge of applying the concepts in an efficient and logical manner. Design patterns are instruments to gather this precious knowledge so that all members of the software community can benefit and more forward.

3.5 Repositories

Design patterns have sparked a surge of information sharing among members of the software community. Buschmann et al write “the pattern form makes it possible to discuss and share knowledge with people who are experts in other domains or even
newcomers and novices in software engineering" [8, p. 418]. The central focus of the discussions is the prose documents describing the solutions. To date, numerous design patterns have been identified in a variety of areas [8, p. 423]. Collections of design form repositories that preserve the existing design solutions.

Popular repositories of software knowledge are libraries. Some software libraries contain collections of commonly used procedures or collections of class hierarchies. One difficulty with these types of repositories is that they are implementation dependent. It would be difficult if not impossible to incorporate entities from repositories written in different languages into one application. Design patterns repositories consist of prose document describing design solutions. Although implementation examples may be given in a specific language, the document describes the solution in terms of the components involved. The notation used in the design patterns may not be the notation to which the developer is accustomed, but the notation used in different object oriented repositories represent solutions using the basic relationships present in object oriented languages. Thus, users of different object oriented languages could understand the solutions described in the pattern. Leaving the implementation to the developers, enables the incorporation of solutions from different repositories into a single application written in one language.

Repositories here refer only to collections of design patterns. There exist several books that present a collection of design patterns. These can be viewed as small repositories. These repositories describe patterns in a defined notation accompanied by discussion in a defined format. Some books have adopted formats from other books. In any case, these smaller repositories represent patterns from many areas and practitioners in
the community can understand the patterns on some level despite their language background. Repositories represent a means of preserving the valuable information available in a community; so, many researchers are working to bring the vast sets of smaller repositories together into a single repository that is readily accessible to all practitioners in the software community.

3.6 Process/Organization

Software design and implementation are key aspects to developing a software system. However, workplace issues also influence the development of a software product. Some adopters of the design pattern concept are applying it to the organizational and process side of software development to make the process more effective and efficient.

Studies into the process and organizational side of software development have revealed patterns. It was discovered that a variety of sound software organizations share similar characteristics. Identification of recurring characteristics and processes that encourages stable software development is a valuable find. As software is in high demand, the need for strong software development organizations is crucial to many industries.

James Coplien and others have focused their research efforts into identifying process and organizational patterns. Coplien wrote his first process pattern in 1993. Since then, he has authored a collection of organizational patterns, including a pattern describing how organizational structure can support software project management [8, p 421]. Steve Berczuk has been credited with creating a pattern language that demonstrates how process and architecture work together [6, p. 39]. Many other researchers have contributed to the body of organizational patterns. Research continues in this area.
In summary: design patterns contribute to many areas of the software development community and to the software community as a whole. It enables the capture of system behavior in revealing the interaction between objects. Design pattern support the concept of abstraction by presenting a general solution without cumbersome implementation details and enables developers to concentrate on arrange multiple solutions into a stable design that facilitates focused development and easier maintenance. They introduce a consistent vocabulary among developers. Design patterns capture valuable design expertise gained by various practitioners and encourages efficient transfer of design expertise among developers. Repositories of design pattern preserve available design knowledge in a form understandable to developers of different language backgrounds. Research in the area of process and organization patterns has identified characteristics and processes exhibited by strong sound software organizations.
4 ADVANTAGES AND DISADVANTAGES

The design pattern movement represents an innovative means of capturing and sharing valuable software solutions among software practitioners. Since the introduction of design patterns, their proliferation through the software community has sparked much discussion and debate. Among the issues being discussed are the advantages and disadvantages of them. Noted names in area of design patterns have cited advantages in software system description and software design [9], [10], [11], [12]. Disadvantages have been cited in the overall novelty of the movement and the area of training [10], [13]. There are advantages and disadvantages the software development process area with the use of design patterns [9], [10].

4.1 Software System Description

A software system is collaboration of components working together to provide a solution. As discussed in the previous chapter, a software system can be viewed at different granular levels. The components at differing levels may be subsystems, objects, etc. Design patterns capture the interaction between system components at a chosen granularity. Capturing the interaction and relationships among system components helps to describe the function being provided by a set of components. Design patterns "describe a system architecture that is broader than an object or class hierarchy"[9, p. 8]. Using design patterns as an architectural tool, "we can describe important structures and relationships at the system level and instruct designers and implementors on how to use
them"[9, p. 9]. As software evolves over time, understanding the underlying architecture helps in efforts to maintain the system's integrity.

4.2 Software Design

During software design, users of design patterns can inject the system with favorable characteristics that will make it more valuable over time. The modular nature of design patterns enables the incorporation of adaptability into the system early in the system development. Design patterns assist developers in designing hooks into the software; software hooks are areas in the software where new software can be easily added without much change to the surrounding modules [10, p. 47]. In addition, they support key principles of object oriented design, for example, separating an interface from its implementation [11, p. 59]. Design patterns define distinct separations between solutions that resemble an interface. Within the documentation, the pattern describes any necessary input structures and describes structures produced by the pattern. In this case, the internal operation of the solution may be viewed as a black box. Designing toward interfaces rather than implementation minimizes the side effects of changes in one module on other modules. The use of interfaces decreases the cost of some fixes and changes later in the system life. Design patterns describe an orderly means of implementing a solution by representing the component, relationships among them, key issues and trade-offs. Understanding the relationships among the components within modules makes it easier to add functionality; understanding the trade-off assists in maintaining the integrity of the design. The applications that incorporate design patterns may be easier to extend while maintaining stability within the system [12, p. 53].

22
4.3 Design Pattern Novelty

The design pattern movement is the first large-scale attempt to formally capture existing expertise in a consistent format. However, the movement is still in its early stages and undergoing rapid change. The novelty of the design pattern movement has been viewed as a disadvantage. Some needed infrastructure is still in developmental stages. Coplien writes that the pattern community has not matured enough to command a standard review committee [13, p. 84]. Issues preventing the formation of such a body deal with deciding on the breadth and depth of knowledge that the committee should represent. Since the introduction of the design pattern movement, practitioners within in the community have authored numerous patterns. The patterns provide solutions to various problems representing expertise in variety of application domains. The increasing volume of patterns requires some sort of organization. Clines states that “Design patterns classification is not yet useful to practitioners” [10, p. 49]. Current classification schemes are useful in determining whether new patterns are truly unique. However, the classifications are not as intuitive to the average practitioner who is learning design patterns.

4.4 Training

Effective use of design pattern requires careful training. “Some design patterns are unnecessarily hard to learn” [10, p. 49]. The solution may consist of subtle implication that is not readily apparent to the novice user. The documentation used to express the pattern may not be as clear as necessary for a novice user to comprehend. Thus, it is
recommended that training proceed first by presenting the design patterns in simplified terms. Then, proceed to present the more general design pattern. Design patterns are many times presented in catalog fashion. The catalog contains a number of independent solutions. However, there is no guide revealing which patterns are necessarily the "right" pattern to be used in a specific situation. Thus, novice users learn by practice which design patterns work in differing situations. Developing the skills to make these determinations takes some time. Thus, there may be a time learning curve involved from the first introduction of design patterns to making effective use of them in applications. Continued mentoring through the initial applications is helpful in providing some explanation and insight that may be initially lacking.

4.5 Software Development Processes

Some software development projects extend over a long period of time and utilize the talents of many individuals. The processes involved in supporting such an effort provide an important infrastructure for the success of the project. "Design patterns coordinate the entire process and community" [10, 47]. Cline [10] writes that design patterns provide a consistent vocabulary that enables communication among practitioners throughout the software's life cycle. Design patterns aid in focusing the design efforts by providing a repository if tried and tested solutions. Cline also states that effective use of design pattern prescribes a course of training for newcomers to the project [10, p. 47]. Maintaining consistency in development within an organization that utilizes design patterns requires consistent training in a uniform set of design patterns throughout the project's life cycle. Design patterns are valuable tools that support the development
process. The capture of existing design expertise “so that inexperienced practitioners can avoid re-inventing the wheel” [9, p. 8]. Design patterns are useful documentation tools. They define a consistent format in capturing solutions and enable more orderly documentation of systems that utilize them. Design patterns can enhance the skills of practitioners using them. For example, object oriented patterns show inexperienced practitioners new object-oriented techniques or other effective techniques to further their existing skills. As design patterns become second nature, practitioners develop greater skills of abstracting problems enabling them to build more robust systems [10, p.47].

“...the benefits of using design patterns may be wasted unless the overall software development process is modified to take design patterns into account” [10, p.49]. The use of design patterns may slightly change the focus of classic processes that support the software development effort. Code reviews for new features should insure that patterns implementing new features maintain the proper intent and usage of the patterns. Changes to an existing implementation should not violate the integrity of the overall design. Design reviews may consist of examining the inter-coordinaton of patterns used in implementing a feature. The impact of design patterns may change the documentation expectations for describing software system. Changes in software process may temporarily upset the current process but with diligence, changing the process to incorporate design patterns may be a change for the better.
5 ACCEPTANCE AND ADOPTION

The design patterns movement has the potential to positively affect the software community. However, achievement of a noticeable impact requires software practitioners to use the concepts and existing patterns. The effect of design patterns in the software community will strengthen as more practitioners accept the concepts and utilize patterns in their everyday development practices. As more practitioners accept and adopt design patterns, the software community as a whole can reap the benefits. Efforts to encourage the acceptance and adoption of design patterns have already begun in organizations through training, pattern mining, and pattern advocacy.

5.1 Training Efforts

Many consulting groups exist that instruct software practitioners in the use of design patterns. Goldfedder and Rising [14] published their experiences during a training program where they instructed a group of software practitioners in using design patterns. The training consisted of two parts: an initial evaluation and a training course.

A variety of practitioners who expressed interest in learning design patterns attended the evaluation meetings. The evaluation meetings were initial attempts to "sell" the concept of design patterns to the group. A copy of the Design Patterns [4] book was given to each practitioner. The book represents tangible evidence of the existence of design patterns. Goldfedder and Rising state that "The patterns sell themselves" [14, p. 61]. The team agreed that design patterns was technology were pursuing and represented a
worthwhile investment. Key observations from the evaluations meeting were teams should learn patterns together to enhance their communication and design knowledge among the members and each member should have a copy of [4].

The course focused on applying design patterns to problems and incorporating their use into the team’s development processes, such as code reviews and documentation. The focus on adopting the design patterns addressed the concerns of helping patterns to last in the organization well beyond the initial training. The course also included discussions of design patterns concepts as well as detailed discussions about the patterns in the design patterns book [4]. The course also consisted of some practical examples using patterns. Although the trainers felt that all of patterns should be covered to exemplify the relationships among the patterns in the book, “Covering all the patterns in a short period of time usually resulted in all the patterns blurring together” [14, 62]. Trainers decided on a longer example divided into smaller exercises. The example incorporated many patterns and some exercises encourage trainees to discover other patterns. “When students begin to use patterns and understand that any pattern is just one piece of the system, they begin to see design at a much higher level” [14, 63]. The amount of implementation, such as languages and actual source code depended on the interest and background of the trainees.

Initially, the course was scheduled for three full days. “An area of some concern was the challenge of most practitioners to absorb the material in just three days” [14, p. 62]. Based on this observation and other observations suggesting that people learn patterns by using them, subsequent course offerings were for “one full day followed by four half days of training” [14, p. 62]. This allowed consulting to be added to the training. During the evenings, trainers help practitioners apply design patterns to their own projects.
The training course was continuously refined through feedback and critiques from the trainers as well as the trainees. Some interesting observations were noted. Open-mindedness is a key characteristic in accepting design patterns concepts. Design patterns represent a change, for some, in the way design is done and for those who are set in their ways, it may be difficult for them to grasp the new approach. Experience in C++ and object-oriented design in not a necessary indicator of ones ability to readily grasp the concepts. “...developers from non-object-oriented project readily grasp the concepts” [14, p. 63]. The demonstration of benefits varied based on the audience. Some were satisfied with sample C++ implementations, while others were interested in the compiler output.

5.2 Pattern Mining

James Coplien’s [13] strategy of getting an organization to accept and adopt design patterns includes an internal focus on the currently existing within the organization. Coplien advocates training organizations in design pattern by encouraging members of the organization to write design patterns. With some guidance, practitioners can learn about design patterns as they develop their own. Those patterns can be used as starting points for the organization’s pattern repository. In addition to writing patterns, Coplien also seeks to discover design patterns already existing within the organization. In [13], Coplien discusses “a pattern technology transfer effort in a large AT&T organization, describing how the organization is starting to embrace patterns” [13, p. 81].

Coplien focused on a group moving from a legacy project to a next-generation project interested in using newer technologies, such as object-oriented programming and
reuse strategies. In amassing support for the pattern effort, Coplien delivered a presentation to managers and engineers about the difficulties of developing a new architecture totally from scratch. He used a familiar example to explain why it was important to take advantage of knowledge gained from the legacy system during the design of the new system. During his presentation, Coplien “offered patterns as way to sustain the corporate memory” [13, p. 82]. Successfully selling the benefits of patterns, he was allowed to begin the process of gathering patterns existing in the organization.

Pattern mining “builds on a major tenet of the pattern discipline: that patterns capture broad practices that have been proven good” [13, p. 81]. The process encompasses finding the patterns, organizing them and publishing them. Coplien proceeded to gather patterns through interviews with key people. Key people were those who understood the system on numerous levels, not necessarily the “politically accomplished experts”. Coplien did not expect the domain experts to speak in patterns. For some, he explained the concepts of patterns and those grasping the pattern concepts made an effort to convey their experiences in pattern form. For others, domain experts who were familiar with patterns were used as interpreters between the pattern miner and the subject being interviewed. Coplien discovered that patterns could be extracted from experts by asking the “right” questions or from the meeting notes at a later time. Coplien also cites HTML as a good way to record patterns because it captures the non-linear reference structure of the patterns through hyperlinks.

After the interviews were completed, the information was examined for the existence of patterns. Coplien lists some criteria used in identifying patterns. The pattern must have been applied in numerous instances with positive results. Coplien cites at least
three as a rule of thumb based on the Biggerstaff’s “Rule of 3” [13, p. 83]. Patterns must maintain positive results over time. For purposes of credibility, the source of the pattern must be a recognized authority in the area of the pattern. Good patterns solve problems indirectly by revealing an underlying behavior model. Patterns describe “the mechanics and arrangements that hold them [the components] together” [13, p. 83]. Patterns tell what to do, not how to do something. The identified patterns are drafted by a person knowledgeable with patterns and reviewed by the expert. Formal reviews require the assembly of a writer’s workshop to further refine the pattern.

5.3 Pattern Advocacy

Continued support of design patterns is essential for an organization to reap the full benefits of design pattern on a long-term basis. After the initial training, an organization must develop strategies to continuously spread the word about patterns and assimilate new patterns into the organization’s repository. Ongoing efforts to adopt patterns into an organization include continued pattern mining, training and consulting.

“Last but certainly not least, it's important to enlist folks from the client organizations as pattern advocates to teach, gather, and organize patterns” [13, p. 84]. Coplien used pattern-mining techniques initially to gather the existing knowledge from a legacy project within the organization. In ongoing support of the pattern effort, a domain expert in the organization accepted the challenge to continue the collection efforts on the legacy project and the new project also developed pattern advocacy strategies.

After training, the current members of the organization have developed skills to use patterns. However, organizations change over time. As new members join the
organization, training them in patterns is means to maintain the effective communication and introduce them to the knowledge existing in the organization. Goldfedder and Rising [14] note that after the initial training, the organization developed an internal training class to spread the word about patterns. The course presented design patterns using existing architectures and included sessions that encourage pattern mining and writing patterns.

Continued consulting helps to reinforce and enhance pattern skills. Goldfedder and Rising state that "training alone without the consulting and hands-on mentoring would not be as effective" [14, p. 64]. Trainers revisited the organization a few times after the initial training for review sessions and to offer additional consulting. Coplien offered to work with members to AT&T organization on continuing basis to refine the pattern mining processes.
6 PATTERN COMMUNITY ACTIVITIES

6.1 History

Over the last decade significant strides have been made to introduce design patterns to the mainstream software community. Beginning as early as 1987, Ward Cunningham and Kent Beck, inspired by the work of Christopher Alexander, presented their patterns for user interfaces – WorkPerTask, FewPanes, StandardPanes, NounsAndVerbs, and ShortMenus – at the Object Oriented Program Systems and Applications (OOPSLA) conference. They presented during Norm Kerth’s workshop entitled “Where objects come from”. Meanwhile, working on a Ph.D. thesis, Erich Gamma “had realized that recurring design structures or patterns were important” [15]. Gamma questioned how to capture and communicate these structures. In early 1989, James Coplien had been cataloging language specific C++ idioms, which were used as teaching aids for objects and C++ at AT&T.

In 1990, at joint conference of the European Conference on Object Oriented Programming (ECOOP) and OOPSLA, Bruce Anderson gave a presentation that was attended by Gamma, Richard Helm, and others. Anderson and some of the attendees discussed the topic of patterns. Later, Gamma and Helm identified a small set of patterns – Composite, Decider, Observer and Container [15]. Some of the patterns identified would later be published in a catalog of patterns. In 1991, “things really got rolling” [15]. Anderson conducted a workshop at that year’s OOPSLA conference that was attended by individuals who would become major contributor to the software patterns community.
Those present included Cunningham, Beck, Helm, Kerth, Richard Johnson, John Vlissides, and Desmond D'Souza. Also, Coplien’s book, *Advanced C++ Styles and Idioms*, was published. At the same time period, Peter Coad, who was discovering the importance of recurring patterns, mentioned the concept recurring patterns in his newsletter.

Anderson repeated his workshop at OOPSLA 1991 conference. By the conclusion of the conference, the Gang of Four (GOF), comprised of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, was formally established. They would publish one of the most noted texts on design patterns [4]. Also in 1991, another group of authors including Frank Buschmann and others published a text on design patterns [8]. Meanwhile, Coad published an article in Communications of the ACM (CACM) about patterns [16].

In 1993, there were many significant events in the pattern community. In May, a group of pattern pioneers including, Vlissides, Anderson, Beck, and D’Souza, facilitated a workshop at IBM where they taught attendees how to use object. However, after hours they discussed patterns. In August, at a Colorado retreat sponsored by Grady Booch and others drew together a group of noted pattern researchers. They converged on the foundations of software patterns. They built upon Gamma’s foundations of software design pattern; however, they called their patterns generative (creational) to distinguish them from Gamma’s patterns and denote the use of these patterns as Alexander had used his patterns. During 1993, the Hillside group, concerned with the use of software design pattern, was formed. At OOPSLA 1993, Anderson held a workshop with the term patterns explicitly listed in title workshop’s title and in the conference agenda. This was
the first time that the term patterns was used at an OOPSLA conference.

In April 1994, the Hillside group met to plan the first Pattern Languages of Programming (PLoP) conference which was held the following August. Also, the book [4] authored by the Gang of Four was published and presented at the OOPSLA 1994 conference. In May 1995, the proceedings of the PLoP conference were published entitled “Pattern Languages of Program Design”.

6.2 Conferences

Much of the proliferation of patterns into the software community has resulted from various discussions among practitioners. The results of those discussions were shared at annual public forums called conferences. Two major conferences – OOPSLA and PLoP - have served as the stage to present pattern discussions and development.

The Association of Computing Machinery (ACM) sponsors the OOPSLA conference. OOPSLA is described among the ACM’s special interest groups on programming languages (SIGPLAN). The ACM SIGPLAN explores programming concepts and tools, focusing on design implementation and efficient use. OOPSLA is a yearly conference held around October. Further information can be obtained from the ACM’s website at http://www.acm.org.

The PLoP conference was developed by the Hillside group to directly address discussions about design patterns and pattern languages. EuroPLop is the European branch of PLoP. PLoP and EuroPLop differ from other conferences in five key areas [8, p.417]. They focus on gathering proven solutions rather than the latest scientific results. The author of a submitted pattern need not be the original developer of the pattern.
Interactive reviews of submissions are performed before the final review by conference chairpersons. At the conference, authors and conference attendees review the pattern submissions, rather than just having authors present the patterns. Also, authors are encouraged to incorporate feedback gained from the conference into submissions before the submission appears in the final conference proceedings. The proceedings of the conference are published as a series of books.

In addition to these conferences, other conferences and workshops exist where patterns discussions have been conducted. Examples of these forums include EuroPLoP, European Conference on Object Oriented Programming (ECOOP) and the International Workshop on Using Patterns (UP).

6.3 Noted Community Figures

Throughout the development of the Design Pattern community, numerous figures have emerged at the forefront of the continued development and support for the community.

The GOF “paved the way for the wide acceptance of patterns in software engineering” [8, p. 415]. They authored one of the most popular design patterns books, Design Patterns: Elements of Reusable Object-Oriented Software.

Erich Gamma got interested in patterns while developing ET++, an extensive class library and application framework in C++. “Currently, he is a Technical Director at Object Technologies International (OTI) where is involved in the development of IBM’s Visual Age” [17].

Richard Helm met Erich Gamma at OOPSLA 1990 conference and realized that
they both "had common ideas about what was important when writing reusable object-oriented software" [18]. He is currently a consultant in object technology with an IBM consulting group.

John Vlissides is a researcher at IBM I.J. Watson Research Center. His research interests include object-oriented design tools and techniques, and frameworks [19]. He also writes the "Pattern Hatching" column in the "C++ Report".

James Coplien among his many works authored C++ Programming Styles and Idioms, which is a widely recognized catalog of C++ good practices. He currently works on developing organizational patterns, which describes the organizational structure of software organization and projects. In addition, he writes a pattern column with Vlissides in the "C++ Report".

The Hillside group is "a nonprofit corporation dedicated to improving human communication about computers by encouraging people to codify common programming and design practices" [20]. It is "The unofficial steering committee of the pattern community" [8, p. 418]. The Hillside group sponsors various conferences including PLoP and EuroPLoP. Members of the group include Ward Cunningham, Desmond D'Souza, and Norm Kerth.

Others who have made significant contributions to the pattern community include [8, p. 415] Peter Coad was one of the first persons to present patterns to the public. He has published a book of patterns concerning domain analysis and using OO techniques to develop applications. Douglas Schmidt has authored many patterns on distribution and high-speed networks used in industrial communication software systems. Robert Martin
describes patterns for use in C++ application. His patterns are considered to be somewhere between patterns and idioms. Wolfgang Pree “has studied pattern as the structural principles for framework principles” [8, p. 416].

6.4 Extended Community Support

In addition to the sharing information in conferences and through the publication of numerous texts about design patterns, the community has other forms of support to further stimulate the sharing and discussions. Articles about design patterns appear in technical journals such as C++ Report, Journal of Object Oriented Programming (JOOP), the Communications of the ACM, and IEEE Software. The Internet provides many websites, and mailing lists focused on design patterns. The Patterns Home Page, http://hillside.net/pattems/patterns.html, “provides useful information about forthcoming pattern events and available books on patterns, and offers references to other Web pages about patterns, such as the Portland Pattern Repository at http://c2.com/ppr” [8, p. 417]. Examples of pattern mailing lists are patterns@cs.uiuc.edu, which discusses patterns that people want to share, and patterns-discussion@cs.uiuc.edu, which discusses issues related to patterns [8, p. 417].
7 SAMPLE APPLICATIONS

This section contains the documentation for the sample applications implementing selected design patterns presented in the Gamma et al book [4]. The selected design patterns are:

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Pattern Type</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Builder</td>
<td>Creational</td>
<td>Object</td>
</tr>
<tr>
<td>2. Composite</td>
<td>Structural</td>
<td>Object</td>
</tr>
<tr>
<td>3. Adapter</td>
<td>Structural</td>
<td>Class</td>
</tr>
<tr>
<td>4. Mediator</td>
<td>Behavioral</td>
<td>Object</td>
</tr>
<tr>
<td>5. Strategy</td>
<td>Behavioral</td>
<td>Object</td>
</tr>
<tr>
<td>6. Template Method</td>
<td>Behavioral</td>
<td>Class</td>
</tr>
<tr>
<td>7. Façade</td>
<td>Structural</td>
<td>Object</td>
</tr>
</tbody>
</table>
7.1 Builder Pattern Application Documentation

**Pattern Name:** Builder

**Intent:** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**Structure:**

![Class Diagram for Builder Pattern](image)

**Figure 1 - Structure Class Diagram for Builder Pattern**

**Discussion:**

The Director contains a reference to a Builder object. The `Construct()` method parses the input structure. For each object in the structure, the appropriate `BuildPart()` method of the Builder object is invoked to perform the construction. The Builder class contains declarations of methods to build all possible parts in the input structure. The ConcreteBuilder inherits all behavior of the Builder class and overrides any of the `BuildPart()` methods with its specialized implementation. The ConcreteBuilder object performs its construction based on the directives of the Director object. When the construction is completed, the Director object directs the ConcreteBuilder object to display its assembled product by invoking its `GetResults()` method.
**Sample Application:** Text Converter

**Problem:** For a given simple HTML file, produce a comparable ASCII formatted file or a scrollable window-based representation containing the LaTeX representation. The user specifies the type of conversion using a command line argument.

**Design Pattern Solution:**
Figure 2 - Class Diagram for HTML Text Converter
Operation:

The application uses the standard MS-DOS command line interface.

The command to execute the application is:

> java HTMLConversionDirector ConversionOption* Filename**

* The ConversionOption parameter specifies the output format of the conversion. Valid values of this parameter are:

ToAscii – converts the text in the given HTML file and saves the output in an output filename whose name is derived from the filename string and the “.txt” extension.

ToLateX – converts the text in the given HTML file and displays the converted text in a temporary window. The window offers two option buttons: The “Save and Close” button saves the displayed text in an output file whose name is derived from the input filename string with the “.tex” extension and terminates the application. The “Close” button terminates the application without saving the displayed text.

** The Filename parameter specifies the HTML filename that will be converted to the chosen output format.

Solution Discussion:

The application chosen to demonstrate the builder pattern is an HTML text converter. It converts an HTML format file into an ASCII format file or LateX format file. The converter recognizes a subset of the valid HTML tags. The converter recognizes the beginning and ending tags for the following: Header, Body, Title, Paragraph, Ordered List, Unordered List, List Item, H1 Font, H2 Font, and H3 Font. The output of the
HTML to ASCII format conversion is saved to text file whose name is derived from the input filename string with the "txt" extension. The output of the HTML to LateX is displayed in a temporary window. The output window offers two option buttons. The option buttons give the user the opportunity to save the displayed text into file, whose name is the input filename string with the "text" extension, or exit without saving the displayed text. During the implementation of this application, observations about Java and design patterns were noted.

Design patterns describe solutions to problems. These solution descriptions can be developed into stand-alone applications. However, complex systems require the integration of multiple solutions into one system providing a more robust application than a single problem solution could provide. In the case of object oriented solutions, integrating solutions requires the merging of multiple inheritance hierarchies. Java imposes a restriction on class inheritance. Although any class can implement any number of interfaces, a single class may inherit implementation from only one base class. The presented implementation of a text converter gives an example of merging inheritance hierarchies. The branch of the hierarchy for the LateXWindowBuilder class must merge the inheritance hierarchy provided by builder pattern and the hierarchy of the output window. An approach by a novice programmer would be to have the LateXWindowBuilder inherit from both the ConvertedHTMLOutputBuilder class and the Frame class. At first glance, this seems easier to let one class inherit from many classes and the user could choose from the available implementations. The disadvantages of this approach are that problems may arise in the form of attribute and method conflicts and it will be harder to extend such a class because it has many different concerns. The
restriction placed on inheritance requires the developer to consider the design of the inheritance hierarchy early in the development cycle.

Java provides a variety of useful utilities that eases implementation. For this application, Java provides such utilities as the StringTokenizer class and the StreamTokenizer class. These classes allow a string and file stream to be treated equally as input streams to be parsed by a set of tokens. These classes eased the parsing of text. In the C++ language, text is parsed character by character and assembled into text strings that are tested for recognition. With Tokenizers, the developer specifies the special character tokens. The tokenizer class handles the expected checking, such as checking for no more token characters and the end of stream. In addition, it provides an easy means of controlling iteration through the chosen input stream and extracting the desired text. These utilities were specifically helpful in this application since a major task in the application was parsing text files.

The builder pattern, as described by Erich et al. [11], was followed according to the description in order to develop this application. The explanation given by a class structure, a description of the relationship between the classes and an example was helpful during the implementation. With most of the design work completed, implementation followed smoothly. The challenge arose in attempting to merge the hierarchy of the design pattern with the hierarchies necessary due to the added requirements of the specific application. Decisions were made as to the class inheritance and containment of other classes. The additional discussion of the consequences and the limitation of the solution were helpful in making important trade-off decisions. They were also helpful in citing potentially unanticipated trade-offs that would normally appear later in the development. For
example, an implementation point was cited as to the use of an abstract interface or concrete class with empty methods as a template for the builder. The choice did not seem significant in early stages of development. However, later during implementation, it was realized that using an abstract interface would require all classes that inherit from the interface to implement all of the methods described. However, in this application the two builders only needed to implement a subset of the methods available. If they received calls for methods that were not supported the default action was to do nothing. Using the concrete class eliminated the need to implement empty methods in the derived builders. This avoids cluttering the derived class with empty functions.
7.2 Composite Pattern Application Documentation

**Composite Pattern**

**Intent:** Compose objects into tree structures to represent part-whole hierarchies allowing clients to treat individual objects and composite objects uniformly.

**Structure:**

![Composite Pattern Structure Diagram]

**Discussion:**

The *Client* contains a reference to a *Component* object. The *Component* object may be a *Leaf* or a *Composite* object. The *Leaf* object is simple object with no references of other *Component* objects. The *Composite* object is an aggregate object that contains a reference to one or more *Component* objects, which may be *Composite* or *Leaf* objects. The client can invoke the same methods on both the *Leaf* and the *Composite* objects because they contain the same methods due their common ancestor. The recursive nature of the structure enables clients to invoke methods in a Composite object and the Composite object can invoke the same method on all of its children.

---

46
**Sample Application:** Dependency Traverser

**Problem:** When building software targets, some targets depend on other targets. Generate composite structures capturing the dependencies among Simple and Composite targets as described in a plain text file.

**Design Pattern Solution:**

![Class Diagram for Dependency Traverser](image)

**Operation:**

The application uses the standard MS-DOS command line interface. The command to execute the application is:

```bash
java DependencyTraverser
```

This application requires no command line arguments, however it does expect to read a file titled “Makefile.txt”. The Makefile defines the targets by name and describes the dependencies among them. The Makefile adheres to the following rules:

1. One target is defined per line
2. A target is declared by its name separated by a colon (:) character followed by the names of the targets that it depends on separated by a comma (,) character.

3. Each line is terminated by a semicolon (;) character.

4. All SimpleTargets are declared before any CompositeTargets.

Below is an example of a "Makefile.txt".

```
Target1:;
Target2:;
Target3:;
Target5:;
Target6:;
Target4: Target1, Target2, Target3;
Target7: Target5, Target6, Target4, Target1;
Target8: Target4;
Target9: Target8, Target7;
```

**Figure 5 – Example of "Makefile.txt"**

**Solution Discussion:**

The application chosen to demonstrate the builder pattern is dependency traverser. This application is a rough approximation of the make process in the Unix environment. The application parses the dependency file, which defines targets and the dependencies among the targets. The application expects a file to be named "Makefile.txt" and adhere to form rules described above. An example of the Makefile is given in Figure 5. For each target, a dependency list is constructed. The application provides two methods: Build and View. The Build method simulates the construction order of all targets. The simulation uses simple print statements to document the path required to build each target as defined in the Makefile. The View methods returns a list of all unique targets that a single target depends on. This informs the user of all targets that must be built in order to construct a specified target. The application automatically exercises these functions on the Makefile
residing in the execution directory. During the implementation of this application, observations about Java and design patterns were noted.

The Composite pattern enables simple and composite objects to be treated in the same manner because they both inherit from the same base class. Thus, the shared interface should be carefully designed. For ease of implementation in this application, the interface for the derived classes is actually a concrete class with empty or trivial implementations. However, as the number of components using the same interface grows, design of a good interface may take some thought because the functions in the interface must be applied to all derived classes. Shared interfaces have some implications. In this application, the declaration of the View method requires the return of an Enumeration. For a Simple target, the View method could simply return the target's name. However, the Composite target contains multiple child targets. This implementation favors the Composite target as it can return an Enumeration of its children. However, this means that the Simple target must create a Vector, add itself to the Vector and return an Enumeration. This implementation requires the Simple target to utilize more memory in order to conform to the interface specification. This is one trade-off that was encountered during implementation. This trade-off has little effect on such a small application however, as applications scale in size, the trade-offs between code size, memory usage, and execution speed become very important.

Java language provides comparable utilities to that of the C++ implementation as given in the Erich et al [4]. For this application, an extension of the Vector class was used as a primary means of the storage. The Vector enables storage of a list of objects. Comparable to the Iterator template in C++, the Vector class implements the
Enumeration Interface. The Enumeration interface enables easy traversal of elements in the Vector. The Vector is comparable structure to a List template in C++. The C++ templates require the user to specify the types of objects that will be stored in the template before it can be used. Java does not track the type of objects added to a Vector. Thus, the user must cast the object to the specific type of object when removing the object from the Vector. Although this is more general, it could present some problems if care is not taken to correctly cast the objects.

For this application, the format of the file was explicitly defined. This application is interested in the file as a set of lines rather than a set of words or characters. Among Java’s numerous input/output streams, there is the LineNumberReader which can read a file object one line at a time. There are also other interesting methods offered by this stream, such as keeping track of the number of lines read and allowing the user to set the line number for a specific line. Although the stream library may seem confusing at first, utilizing the different types of stream introduces developers to the power of the Java streams.

Since many real life objects can be expressed in the form of part-whole relations, Composite structures are very common. For example, drawings are composed of simple objects (e.g., circles, squares, etc.) and composite objects (e.g., frames containing simple objects). Computers are composed of subsystems that consist of smaller subsystems of individual parts. The application of composite structures occurs in many different industries utilizing software to track relationships between different parts. The wide application also means that there are many trade-offs that can be made during design and implementation. Erich et al [4] presents nine issues that are recommended during
implementation. Other issues were discussed above. For small applications, some of the issues may not be directly applicable; however, they provide good insights as applications scale upward in size and complexity.
7.3 Adapter Pattern Application Documentation

**Adapter Pattern**

**Intent:** Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that otherwise couldn’t because of incompatible interfaces.

**Structure:**

Object:

![Figure 6 - Structure Class Diagram of the Adapter Pattern](image)

**Discussion:**

The client contains a reference to a *Target* object. The *Target* object represents the interface that the client wants to use. The *Adapter* class inherits the interface and behavior of the *Target* class. The *Adapter* contains a reference to the *Adaptee* object. The *Adaptee* object has an interface that is not directly compatible with the *Target* object’s interface. In an effort to allow the client to utilize a chosen interface to access the behavior of an incompatible interface, the *Adapter* object overrides the behavior that it inherited and forwards specific requests to methods in the *Adaptee* object. The client believes that he is still interacting with the usual *Target* object; however, the *Adaptee* object supplies the specific behavior for the *Target* object.
Sample Application: Image Button Adapter

Problem: Java language is noted for its ease of programming interactive elements in applications. Java defines a protocol for handling event driven interaction. The release of Java 1.1 introduced changes in that protocol from the Java 1.0. Any applications still using the Java 1.0 protocol were suggested to revert to the new event handling protocol.

Buttons are popular interactive elements whose event handling protocol was affected by the new release. The objective of this application is to adapt the event handling protocol of a Java 1.0 button object to the Java 1.1 event handling protocol without changing the Java 1.0 event handling interface that the Java 1.0 button is accustomed to using.

Design Pattern Solution:

![Class Diagram for Image Button Adapter](image)

Figure 7 – Class Diagram for Image Button Adapter

Operation:

The application uses the standard MS-DOS command line interface. The command to execute the application is:
This application requires no command line arguments. This application displays a panel with a button located in the center of the panel. Instead of labeling the button with a string, the button's label is an image. The image is specified at creation by supplying the constructor with the name of the image file. The application supplies the constructor with the name of .gif file located in the application directory. If no image label is supplied, the application uses a default image file – orb.gif, which is also located in the application directory. When the user clicks the button using the mouse, an exit message is printed to the screen and the application is terminated.

**Solution Discussion:**

The application chosen to demonstrate the adapter pattern is an Image Button Adapter. This application adapts the event handling protocol of a Java 1.0 button object to the protocol used in the Java 1.1. This sort of application is useful since buttons are popular interactive elements. Java 1.0 applications using buttons need not change drastically in order to take advantage of the new event handling protocol. The application displays a panel with a button in the center. The user interacts with the button in order to terminate the application. The ImageButton uses the Java 1.0 event handling. Interaction with the button generates events in the Java 1.0 format. The panel, which behaves as a container for the button, handles events according to the 1.1 protocol. In order for this adaptation to proceed, the Java 1.1 button listeners actually register with the panel and listen to the panel. The panel receives all of the Java 1.0 button events, translates them to the Java 1.1 format and notifies its listeners that a button event has been generated. The panel notes itself as the source the event. The listeners view the container as the actual
object. The listeners perform actions defined in their actionPerformed method in response to the notification. In this application, there is only one listener; however, more listeners could be added using the addActionListener method. The panel is allowed to handle the Java 1.0 button events because the button does not define a method to handle the events generated by the button and by default, events not handled by the components are passed their parent containers.

Event handling is key aspect of the Java language. Although a Java application can be programmed to execute without user intervention, some applications require user interaction. User interfaces can be in the form of utilizing the mouse to make selections, moving the mouse around the screen or entering text in designated locations. All user interactions with Java application generate some sort of event. The Java event handling protocol allows the application to react or respond to the user events. Applications respond in many ways including opening new windows, terminating themselves, and saving information supplied by the user. Since event handling is such an important part of Java. A change to the protocol has the potential to affect many applications.

In the Java 1.0 event handling protocol, the components, such as buttons, text areas, etc., were responsible for handling their own events. This was deemed as very inefficient since it required the developer to subclass an element, such as button, in order to implement the event handling. Also, all classes extending that button must provide event handling if it is different from the parent. In the Java 1.1 protocol, object listeners register with the objects that they want to listen to. Any object can be an action listener as long as it implements the appropriate listener interface. When objects generate events, the listeners are notified and the listeners perform some specific action in response to the
event. This decouples the interactive objects from their event handling. This is sufficient in most cases because this protocol allows more than one listener to respond to the same event.

It should be noted that as in the Java 1.0 protocol if an object did not handle its own events and the parent did not handle the child’s events, then no action was taken in response to the event. This situation can happen in the Java 1.1 protocol. If there is not at least one listener for an object or event, then no action is taken in response to the event. An application may not handle all events generated within the application, so this situation may not present a problem. However, failure to handle a necessary event may present a problem.

The adapter pattern can be used with classes or objects. Erich et al [4] notes that implementation of the adapter pattern for classes and objects has different consequences. This implementation focused on application of the adapter pattern to objects. One of the consequences of the object adapter is that a single adapter can work with many adaptees. In this application, there was only one button. However, in other applications, there may be multiple interactive objects on one panel. So this application could be extended to allow this panel to collect events for multiple Java 1.0 objects that may reside in the panel and have multiple listeners only interested in certain events.
Although the above consequence can be applied directly to a Java application, Erich cites another issue—"using two-way adapters to provide transparency" [4, p. 143]—which may not be implemented in Java with its current limitation on inheritance. This issue presents a solution where an adapter inherits from multiple classes in order to provide transparency when information is required to flow back and forth through the adapter. This issue may have a different format when applied to a Java class hierarchy.
7.4 Mediator Pattern Application Documentation

**Mediator Pattern**

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

**Structure:**

![Structure Diagram for Mediator Pattern](image)

**Discussion:**

The *Colleague* class represents the base class of objects that must communicate with one another. The *Colleague* object describes the common interface for all of its descendents. It also contains a reference to a Mediator object that it will depend on to provide its communication management. *ConcreteColleague1* and *ConcreteColleague2* classes inherit the *Colleague* interface. Each concrete class implements its specialized behavior. The *Mediator* object “defines the interface for communicating with the *Colleague* objects” [4, p. 277]. The *ConcreteMediator* inherits and implements the communications described by the Mediator class. The *ConcreteMediator* contains references to all of the *ConcreteColleague* classes that it must manage communications between.

**Sample Application:** Font Display

**Problem:** Text can be displayed in a variety of font styles. Develop a Graphical User
Interface (GUI) that allows a user to view a string of text in various font styles. The font style should include a specific font type, a font weight (plain or bold), a font slant (plain or italic), and a font size (in points).

**Design Pattern Solution:**

![Class Diagram for Font Displayer Applet]

**Operation:**

The application uses the Java Applet Viewer tool that accompanies the Java Development Kit (JDK). The appletviewer allows developers to view HTML files. It serves as a development alternative to a fully functional browser. Applets that can be viewed using the Applet Viewer can also be viewed in a Java enabled browser.
This application is executed by displaying the appropriate HTML file using the appletviewer. The HTML file includes the compiled application file, whose name is the applet name with the .class extension. The application uses the standard MS-DOS command line interface. The command to execute the application is:

```
> appletviewer MediatorApp.html
```

The application displays a GUI window with default settings already selected (See figure 10). The object of the application is to display the displayed text string in a variety of font styles. The user interacts with the application by using the mouse to make selections in the window. The user can select the font type from the list. The user can select whether the string is bold or plain, or italicized by selected the correct checkbox. The size of the font is selected by using the scrollbar. The available font sizes are displayed in the size text field. The available font sizes are 12, 24, 36, 48. When the user has selected the desired font style parameters, selecting the “Apply” button will change the displayed text string to desired font style. Selecting the “Exit” button will end the application.
The quick brown fox...

Family: Courier

Weight: plain, bold

Slant: plain, italic

Size (point): 48

Figure 10 – MediatorApp GUI Window

Solution Discussion:

The application chosen to demonstrate the mediator pattern is a Font Displayer applet. The Font Displayer uses a graphical user interface to let a user select parameters of font style to display a string of text (See Figure 10). GUIs are typical configurations in which many components must communicate with one another. Communication between the components becomes crucial because the user interaction with components can occur in any order. Some components may require knowledge of more than one component's state change.

The Java class library contains many popular GUI components, such as text areas, buttons, and checkboxes. Each component extends the behavior of the Component class adding methods to achieve a specialized behavior. Figure 9 shows a direct inheritance
relationship between Component and the derived GUI components. It should be noted that some GUI components do not inherit directly from the Component class; there are intermediate classes in the inheritance chain between the Component class and the GUI component. An example is the TextField class. The availability of such a variety of GUI components eases the development of fully functional GUIs. Creation of a specific component requires only instantiating the class. The core behavior is already present; however, overriding methods can further customize a component's behavior.

GUI components generate events when a user interacts with them. As mentioned in the Image Adapter Button application discussion (above), the Java 1.1 event-handling scheme has changed from the Java 1.0 method. Java 1.1 requires listeners to be registered to respond to events. Since the MediatorApp is coordinating the interaction between the components, it listens to events generated by the components. The MediatorApp is notified of events by implementing the interfaces (ItemListener, AdjustmentListener, and ActionListener) and registering with the object whose events are collected by the interfaces. The MediatorApp behaves as an observer to components in the GUI. The Gamma book [4] contains an observer pattern. In contrast, Java implements the observer objects through the implementation of interfaces used to collect event information.

Although Java only allows a class to inherit implementation from one base class, a class can implement many interfaces. Note that the MediatorApp only inherits implementation from Applet class. The use of interfaces has subtle implications. Implementing an interface requires the class to provide implementation for the abstract method(s) described in the interface. If the class fails to provide implementation, the derived class becomes abstract and can not be instantiated. When a class provides
implementation for an interface, the class can be used anywhere the interface can be used. In a sense, implementing an interface gives the class an alias name. For example, the Button component requires an ActionListener object to be registered with it in order to listen to its events. Since the MediatorApp class implements the ActionListener interface it can used when an ActionListener object is expected.

Some consequences and implementation tips given for this pattern represent tradeoffs that are neither inherently good or bad; they are heavily dependent on the user's implementation requirements. One main focus of the trade-offs occurs when considering easier management with a little more complexity in a centralized area. Since the complex interaction among the components are captured in one place (the Mediator class), changing the interaction means changes to one class. In addition, the complexity of the Mediator class allows the code for the components to remain relatively simple. Another focus of the implementation notes is extensibility. Decoupling the objects from their interaction allows the components and the Mediator to be more reusable than if all the components contained references to one another. Also, the use of an abstract Mediator class allows the same group of components to work with different Mediators. During the implementation, it was decided to omit the abstract Mediator. However, simply subclassing the MediatorApp class and overriding the methods can also result in the same components interacting in a different way. It is realized that this choice has drawbacks as well, such as the opportunity for subclassing is limited with the current implementation.

In any case, the implementation of this pattern in Java was straightforward. Java also gave some assistance by predefining popular components and their behavior. The consequences cited were helpful as guides for avoiding some unnecessary complexity.
7.5 Strategy Pattern Application Documentation

Strategy Pattern

**Intent:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**Structure:**

![Figure 11 – Structure Class Diagram for Strategy Pattern](image)

**Discussion:**

The *Context* contains a reference to a *Strategy* object. The *Strategy* class represents the interface that all *ConcreteStrategy* classes will follow. The *Context* class will use the *Strategy* class interface to communicate with a *ConcreteStrategy* object. The *ConcreteStrategy* classes inherit the interface of the *Strategy* class and implement the methods in the interface according to its defined algorithm.

**Sample Application:** Graph Manipulator

**Problem:** Given the description of an acyclic directed graph define two strategies to represent the graph and view the nodes in it.
Design Pattern Solution:

![Diagram of DependencyTraverser, GraphStrategy, and GraphNode classes with methods such as addNode, connectNodes, viewNeighbors, and viewAllDescendents.][1]

**Figure 12 – Class Diagram for Graph Manipulator**

Operation:

The application uses the standard MS-DOS command line interface. The application expects to read a file titled “Makefile.txt”. The Makefile defines the nodes by name and describes the dependencies among them. See the operation section for Composite application for format and specification of the Makefile. The command to execute the application is:

> java DependencyTraverser RepresentationOption*

* The RepresentationOption parameter specifies the format in which the graph will be stored during the execution of the application. Valid values of this parameter are:

   - **List** – The graph is stored as a Hashtable of lists. Each list contains the dependents of a specified node. The keys in the Hasstable represent the unique nodes defined in the Makefile.txt file. This method is best used with sparsely populated graphs; i.e. graphs whose nodes have a small number of dependents.

   - **BitSet** – The graph is stored in a table and a vector of nodes. The table describes...
the connections between the nodes in the graph by setting a bit in the dependency table to true if there is connection between two nodes. For example, if the intersection between x and y columns of the table is set to true, then node x is connected to node y. The vector of nodes serves to translate the connection descriptions back to the distinct nodes described in the Makefile.txt file. This method is best used with densely populated graphs – graphs whose nodes may have a large number of connected nodes – because space is minimized by storing the actual nodes only once and using bits to represent connections.

Solution Discussion:

The application chosen to demonstrate the Strategy pattern is a Graph Manipulation application. The graph described by a Makefile can be represented in two distinct formats – a list of vectors or two-dimensional connection table. To exercise the storage options the application accesses the storage scheme in order to display the children of a specified node or all of the descendents of a specified node. The output generated by the application is a list of the specified nodes’ descendents displayed in the command window.

The DependencyTraverer is created with a reference to a ConcreteStrategy object – either a ListGraphStrategy or BitSetGraphStrategy. The DependencyTraverer parses the Makefile one line at a time identifying a new node and its dependents if it has any. The DependencyTraverer then invokes add() method for the strategy so that the node can be added to the current storage scheme. This is the same construction algorithm used in the Composite pattern application (see above section 4.2). Once the parsing is completed
either of the view methods – `viewNeighbors()` or `viewAllDescendants()` – may be invoked to list the dependents of a specified node.

This application is good example of how the concepts of multiple patterns can be used together to develop one application. During the development of the application, there was some confusion when it was realized that the basic structure of this application closely resembled the structure of the builder pattern. The object of this application is to exercise two different storage strategies; however, in order to exercise them, the schemes must be constructed and populated. The concepts of the builder pattern were used during the construction and population of the storage scheme. The strategy pattern allowed for the two different storage schemes to exist without requiring any changes to the building scheme. Although concepts of two patterns were being used together it was necessary to understand each pattern’s purpose in the application to alleviate confusion.

Directed graphs are popular computer science constructs. The concepts of representing a set of entities and the dependency relationships among them can be applied to numerous areas, e.g. route planning, genealogy, etc. Graphs may be represented in many different ways. Many times, the method used to represent a directed graph depends on the application requirements and the entities within the graph.

The two strategies used to represent the graph in this application have advantages depending on the graph being used. For a small graph whose nodes have a small number of dependents, the list strategy is most advantageous. The List strategy uses a Hashtable of vectors to represent a group of lists. Each list represents a node along with its list of dependents. The keys in the Hashtable are the names of the unique nodes declared in the Makefile. Since nodes can have dependents in common, a node may appear in the
dependency list of more that one node. Since this is such a simple scheme, common
nodes are actually duplicated. The duplication of common nodes is not detrimental for
small graphs, however, as the number of unique nodes grows and the number of common
nodes grows, duplication may represent a waste of valuable resources.

The BitSet strategy is a more suitable scheme for large graphs whose nodes may
have a large number of dependents. The BitSet strategy stores the actual nodes in one list
and uses an array of bits to represent the dependencies among them. There is no
duplication; each node appears in the list exactly once. If node x is dependent on node y,
then the intersection of the columns x and y in the bit table is set to true. The nodes in the
list are stored such that the row index in the bit table is the same as the index for that node
in the list. This scheme is more advantageous for large graphs because, the connection
table is represented in bits, which consume less space than complete objects, which may
contain references to other objects.

The phrase "large number" has been used many times in this discussion. In
reality, the phrase is relative and dependent on several factors, such as the available
memory, execution environment, etc. The usefulness of the strategy pattern is that the
client can choose the most suitable storage scheme for the application at run-time.
Although, all of the schemes exist as descriptions in the class hierarchy, the actual storage
space is not used until an object of that class type is created. In addition to run-time
selection of the storage strategy, more strategies can be added without changing any of the
other classes in the hierarchy. New strategies can be added virtually unnoticedsd to other
strategies as long as it follows the Strategy class interface. The client should be made aware
that another strategy option exists.
During the initial discussion of this application, it seemed that the Composite pattern application could be extended to include the List and the BitSet strategies for representing the targets and their dependencies. However, it was discovered that this might not work as intended. The conflicts occurred between the strategy application requirements and the subtleties of Composite pattern. The Strategy pattern application required that each graph strategy maintain dependencies of the entities described in the Makefile according to its specific algorithm. The list strategy maintained dependencies in lists. The BitSet strategy maintained the dependencies in a two-dimensional bit array. An implication of the Composite pattern is that a composite object maintains references to its children. Thus, the Composite pattern mandates a specific method of storing and managing a node's children that may be in conflict with the strategy’s method of maintaining dependencies between a node and its dependencies. Thus, it was decided to use a simpler entity to manager – the GraphNode. In general, it was easier to use strategy pattern on loosely coupled entities that pose little or no conflict with the strategy’s method of accessing and manipulating them.

Understanding and deciding which pattern is most applicable in different situations may be difficult at times, because there is no guidance to the “correct” pattern if the developer is unsure of which design pattern is best for the particular situation. It was revealed that other pattern users have had similar experiences. When situations like this occurred, consultation was sought from a more experienced pattern user.
7.6 Template Method Pattern Application Documentation

**Template Method Pattern**

**Intent:** Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms basic structure.

**Structure:**

![Structure Class Diagram for Template Method Pattern](image)

**Discussion:**

The `AbstractClass` contains the declaration and implementation of a `TemplateMethod` along with abstract declaration of primitive operations (`PrimitiveOperation1` and `PrimitiveOperation2`). The `TemplateMethod` specifies an algorithm during which the primitive operations are invoked. `ConcreteClasses` inherit all of the method declaration from the `Abstract Class`. The `ConcreteClass` must override the primitive operations because no implementation is provided from them in the `AbstractClass`. Overriding the primitive operation by the `ConcreteClass` changes the behavior of the functions when invoked during the `TemplateMethod` algorithm without changing the structure of the algorithm specified in the `TemplateMethod`. 
Sample Application: Shortest Pathfinder

Problem: Given the description of an acyclic directed graph and two strategies to represent the graph, determine the shortest path between two nodes in the graph for a selected representation.

Design Pattern Solution:

![Class Diagram for Shortest Pathfinder](image)

Operation:

The application uses the standard MS-DOS command line interface. The application expects to read a file titled “Makefile.txt”. The Makefile defines the nodes by name and describes the dependencies among them. See the operation section for Composite application for format and specification of the Makefile. The command to execute the application is:

```
> java DependencyTraverser RepresentationOption*
```

* The RepresentationOption parameter specifies the format in which the graph will be stored during the execution of the application. Valid values of this parameter are:

- List – The graph is stored as a Hashtable of lists. Each list contains the
dependents of a specified node. The keys in the Hashtable represent the unique nodes defined in the Makefile.txt file. This method is best used with sparsely populated graphs; i.e. graphs whose nodes have a small number of dependents.

**BitSet** – The graph is stored in a table and a vector of nodes. The table describes the connections between the nodes in the graph by setting a bit in the dependency table to true if there is connection between two nodes. For example, if the intersection between x and y columns of the table is set to true, then node x is connected to node y. The vector of nodes serves to translate the connection descriptions back to the distinct nodes described in the Makefile.txt file. This method is best used with densely populated graphs – graphs whose nodes may have a large number of connected nodes – because space is minimized by storing the actual nodes only once and using bits to represent connections.

**Solution Discussion:**

The application chosen to demonstrate the Template Method patterns is the Shortest Pathfinder application. This application extends the Graph Manipulation application used to demonstrate the Strategy pattern. The Shortest Pathfinder application utilizes the two different graph representation schemes (BitSet and Linked List). The user specifies the desired representation as a command line option. The Makefile describing the graph is parsed and stored in the chosen format. The shortestPath method finds the shortest path between a source node and a destination node for the chosen graph representation.

In extending the Graph Manipulation application, there were some cosmetic
changes and functional changes. The cosmetic changes include changing the names of the
\textit{GraphStrategy} class to the \textit{Graph} class and eliminating the \texttt{viewAllDescendents} method, which was not essential to implementation of this application. The function changes imposed new restrictions on the class hierarchy. The \textit{Graph} class is defined as an abstract class and declares three abstract methods. The declaration of the class as abstract prohibits a client from creating an instance of the \textit{Graph} class; the client must instantiate one of the concrete derived classes of \textit{Graph} class (\texttt{BitSetGraph} or \texttt{ListGraph}). The declaration of the abstract functions requires all derived classes to implement the abstract functions or they, too, will be considered abstract. During the implementation, the \texttt{addNode} and \texttt{connectNodes} methods were declared as abstract, however, that was purely a decision made at design time. They could have been defined in the \textit{Graph} class as an empty function that derived classes could override. These methods are a means of getting information into the chosen graph representation; however, the template method does not depend on these methods.

Although a breadth-first search locates the shortest path with the fewest number of searches, the implementation of such a search is more complicated. Thus, the template method, \texttt{shortestPath}, defines the algorithm for performing a depth-first search to find the shortest path between a source node and a destination node. The \texttt{shortestPath} method invokes the \texttt{viewNieghbors} method in order to access nodes on the graph. The algorithm starts by locating the source node in the graph. Then, investigation proceeds by examining each neighbor of the source and each neighbor of the source’s neighbors and so on until the destination is located. During the investigation, a variable keeps track of the path with the smallest number of nodes. After searching all paths from the source node, the shortest
path vector is returned. If the destination node can not be reached from the source node, an empty vector, whose length is zero, is returned.

Gamma et al [4] cite implementation issues in the area of access control and algorithm design. C++ and Java provide means of controlling access to the primitive operations used by the template method; the primitive operations can be declared as protected members. The defining class and subclasses can only access protected members. This is an important issue if the primitive operations change the subclass’s data members. Use of access control can prevent the unintended corruption of class data. Another issue with access control is controlling derived classes’ ability to override the template method. This is the issue that is very easy to forget. While concentrating on ensuring that the derived classes implement the abstract methods, one can forget that the derived class can also attempt to override the template method. It is suggested that the Template method be declared as non-virtual. In [4], it is stated that “An important design goal is to minimize the number of primitive operations that a subclass must override”. The shortestPath template method only utilizes one primitive operation. Derived Graph classes need only implement the viewNeighbor method in order to add functionality to the template method. Good design may result in algorithms that utilize a minimal subset of methods that must be overridden by the subclass.
7.7 Façade Pattern Application Documentation

Facade Pattern

**Intent:** Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher level interface that makes the subsystem easier to use.

**Structure:**

![Figure 15 - Structure Class Diagram for Façade Pattern](image)

**Discussion:**

The Façade class presents a simple interface that hides the complex interactions of the underlying subsystem classes. The Façade class passes requests made by a client to the appropriate subsystem class. The subsystem classes provide specialized functionality that implements the systems behavior. Subsystem classes are not aware of the Façade. Clients need only to comprehend the interface provided by the Façade in order to utilize the functionality provided by the underlying subsystem classes.
Sample Application: Graph Request Handler

Problem: Given the description of an acyclic directed graph, develop a graphical user interface that allows a user to request a list of all paths and the shortest path between any two nodes in the graph.
Design Pattern Solution:

Figure 16 – Class Diagram for Graph Request Handler
Operation:

The application uses the standard MS-DOS command line interface. The application expects to read a file titled “Makefile.txt”. The Makefile defines the nodes in the graph by name and describes the dependencies among them. See the operation section for Composite application for format and specification of the Makefile. The graph represented in the Makefile is parsed and stored as a Hashtable of linked lists. The keys in the Hashtable represent the unique nodes defined in the Makefile.txt file. The value component of each Hashtable pair is a linked list that contains the dependents of the node specified in the key. This storage method is best used with sparsely populated graphs; i.e. graphs whose nodes have a small number of dependents. The command to execute the application is:

```
    java GraphFacade
```

The application displays the GUI window shown in Figure 17. The GUI gathers the user’s request and its parameters. The output of the request is displayed in the command window used to start the application. The user interacts with the application by using the mouse to make selections in the window. The user selects the request – All Paths Request, Shortest Path Request, or Exit - by selecting the correct checkbox. The user can select the desired source and destination strings from the lists located next to the textfield. The lists provide all valid choices for source and destination strings. The Exit Request does not require the specification of source or destination string. When the user has selected the desired request and its parameters, selecting the “OK” button will cause the output generated by the request to be printed in the command window.

78
Solution Discussion:

The application chosen to demonstrate the Façade pattern is the Graph Request Handler application. This application utilizes concepts from the Graph Manipulator application (see Strategy Pattern Documentation) and the Shortest Pathfinder application (see Template Method Pattern Documentation). This application uses the same parsing algorithm to extract the graph description from the input file and store it in the List format. The List format was chosen as a specific example; however, the design could be easily expanded to support other graph representation strategies. The Graph class defines two template methods – ShortestPath and AllPaths. The ListGraph class implements the primitive operations required by the template methods.

The GraphFacade class defines one public method – getRequest - available to clients. This one call coordinates interactions among many of the subsystem classes. The getRequest method causes the GUI to be displayed for the user. When the user selects “OK” in the GUI, the information for the request is gathered and a new request object is created. The new Request object is returned to the user.
The Facade minimizes the coupling and communication between the client and the subsystem classes. Coupling between the client and the subsystem is loosened because the client need only utilize the interface in order to interact with the subsystem classes. Thus, the underlying subsystem classes could be changed without affecting the clients. The Façade masks some of interaction between the subsystem thus the client does not have to communicate with as many objects in the subsystems.

The Facade provides an interface to several underlying subsystem classes, but it does not prevent access to the underlying subsystem classes. Placing methods for all requests to any of the subsystem classes in the Facade interface would result in a very cluttered and cumbersome interface. As discussed above, getRequest method only hides the interactions between the UserInputGUI and the Graph object. However, returning the Request object to the client allows the client to interact directly with the Request object. The GraphFacade object is not concerned with forwarding requests to the methods of the Request object. This implementation allows clients to use the interface to hide some interactions and use some of the subsystem classes directly.

In [4], the Mediator pattern is listed as a related pattern of the Façade pattern. The Façade and the Mediator provide similar function with some subtle differences. Both provide a mechanism that coordinates interaction between the classes of which it is aware. However, the Mediator pattern coordinates the communication between the classes so that they will not directly refer to one another. This imposes a requirement that the classes know about the Mediator. The Façade hides some of the complex interactions without adding any new functionality to the subsystem. The subsystem class beneath the Façade need not know about the Façade.
As a general rule for interface design, an interface should provide clients with useful methods for interacting with the subsystem, but should not be congested with superfluous information. Design of stable interfaces early in design helps to minimize the need for costly changes later in design. Changes to interfaces later in design may propagate to clients that use the interface.
8 CONCLUSION

8.1 Theory Summation

A software system is the product of the software development process. The software development process describes the manner in which customer requirements are gathered and the resulting software system is developed. Two popular models describing the software development process are the Waterfall and the Spiral Model. These models aid in planning software development projects.

Novice developers are introduced to the process with limited training and experience. In an effort to harness existing knowledge about various software systems and transfer that to members of the software community, researchers have looked to architecture engineering and adopted Alexander Christopher's notion of patterns, which describe solutions to recurring design problems. In software, design patterns can positively influence the design phase of the software development process that in turn may positively impact the other phases of the software development process by positively affecting the developers participating in the process.

Design patterns have made contributions to the software community. They provide a description of interaction between objects in a software system. They introduce a layer abstraction to avoid impeding design with overwhelming detail. They provide a consistent vocabulary among co-developers. They capture existing design expertise. Collections of design patterns represent repositories of existing design knowledge. Research in the area of process and organization patterns has identified characteristics and
processes exhibited by strong sound software organizations.

There are advantages and disadvantages in using design patterns. Design patterns can help developers describe software system at varying granular levels. Design patterns can aid software developers in designing more robust systems. Design patterns is a relatively new concept to the software community so the standardizations and organization have not been full implemented. Careful training is required in order to utilize design patterns effectively. An organization’s software development process must change to take in to account the user of patterns.

Efforts are being made to help software developers understand patterns and encourage them to use patterns in their everyday work. Software developers are being trained in patterns using the GOF book [4]. Pattern mining endeavors to extract patterns from existing software systems. On going training and consultation in the form of Pattern Advocacy groups are essential to organizations wishing to continuously utilize patterns in their development environments.

There is vigorous activity in the pattern community. A brief history of the design pattern movement in software was presented. Design patterns are discussed at noted software conferences, such as OOPLSA and PLoP. Noted figures in the design patterns community, such as the Gang of Four and the Hillside Group were identified. The design pattern community has extended support through conferences, workshops, published texts, and numerous websites.

8.2 Java vs. C++ Implementation

Several specific language specific issues were noted between the Java and C++
implementations of the patterns. Java unlike C++ has an inheritance limitation. Java only allows a class to inherit implementation from one base class; however, it can implement any number of interfaces. This limitation was encountered in the Builder pattern where one of concrete builder classes needed functionality from its base class and from a window subclass. At first, the inheritance limitation appeared to be a problem. With a more careful consideration, a decision was made to have the concrete subclass contain a reference to the window subclass. This design turned out to be cleaner and more robust, as it allows the window to be changed without much change to the builder class. Thus, the inheritance limitation in Java forces the designer to consider inheritance relationships more carefully and earlier in the design phase. The resulting design may be better structured and flexible.

Unlike C++, Java methods are type virtual by default. So, subclasses can override any method declared in parent if they wish. However, Java has a “final” method type. The final method type allows the developer to control a subclass’s ability to override methods in its base class. Methods of type final cannot be overridden in subclasses. In the Template Method pattern, the template method, which is an outline of a shortest path algorithm, should not be overridden by subclasses. In the discussion in [4], it is recommended that developers use a naming convention to alert users of the existing template method. However, the use of the final method type prevents subclasses from overriding the template method accidently or intentionally.

All objects in Java are references. Thus, there are no pointers as in C++. This can be viewed as an asset or detriment. It is useful because the developer does not have to perform any dereferencing to get the contents of the objects. However, this requires the developer to carefully consider issues of object ownership. In the Strategy pattern
application, a vector of GraphNode objects is returned to the caller as the result of a search operation. If a reference to the vector is returned to the caller and the caller alters the vector, the result of the alteration will appear in the vector contained in the caller and the callee because the references refer to the same vector object. In this case, it is up to the callee class to insure that the vector is copied to a new vector and the reference to the new vector is returned. Then, the caller solely owns the new vector. Failure to consider the implications presented by references may result in the unintended corruption of data.

All classes in Java are subclasses of the Object class, even user-defined classes. The Object class defines several methods useful to all classes, such as a method to define equality. So, a user-defined class has numerous useful methods already declared for it. The user-defined class can override methods declared in the Object class to provide specialized or extended behavior. This is helpful to novice developers who lack extensive coding experience.

Java provides similar library utilities to C++ with a slightly different look and feel. Java contains a dedicated String class that defines several useful operations for manipulating strings and performing conversions between different string formats. C++ represents strings using a combination of an intrinsic data type, char* or char array, and provides some methods for manipulating those strings in a separate string library. The Java Input/Output (I/O) structure is more complex than the C++ I/O structure. Both contain a dedicated stream for input and another for output. However, the Java I/O stream class hierarchy is more extensive and allows nesting of streams to achieve specialized stream formats. For example, the Java I/O structure provides a stream class to parse files by lines of text. Using this feature with the StringTokenizer class, which can
separate lines of text into multiple subunits based on chosen separator tokens, made the parsing of the HTML file in the Builder pattern application easier.

In addition, Java and C++ differ in their implementation of templates. C++ requires developers to specify the type of objects in the template upon creation. Java template structures, e.g. Vector, do not expect specific type of objects. It is left to developer to know what object was placed in the structure and the object must be type cast when it is removed. Java does provide some safety in that an exception will be generated if an attempt to cast a class to the wrong class is made. With careful casting, Java structures, such as Vectors behave similar to the C++ templates.

In the final analysis, using the Java language did not present any significant barrier to implementing the chosen design patterns. Thus, the Java implementation presented in the attached appendix supports the GOF claim that the design patterns presented in [4] may be implemented in various object-oriented languages.

8.3 Next Steps

Much work has been done to make the software community aware of the importance of design patterns in software. However, there is still much work to do. Some areas that still need further attention include pattern mining, patterns organization, and methods and tools.

Pattern mining is process of gathering patterns, either discovering new patterns or extracting patterns from existing software systems. “Although a lot of patterns are already available, of all scales and degrees of abstraction and for many domains, mining new patterns will remain an important activity for the future” [8, p. 420]. Undiscovered
patterns are being sought in areas including specific application domains and organizational processes.

Many patterns have been discovered thus far. They have been published in a wide variety of mediums including textbooks and conference proceedings. However, the community must endeavor to organize the patterns in a way that is useful to the community. Attempts have been made to organize the patterns into pattern languages and a large pattern map, which shows the relationships of a group of patterns. The Hillside group is working on a system where units called “pattlets” are linked together using several different relationship types. A pattlet is a brief abstract of a design pattern. [8, p. 423]

The design patterns presented in the “Gang of Four” book [4] are presented as isolated patterns. However, in developing software system, one may wish to incorporate numerous patterns. Efforts are underway to create tools that ease the use of patterns. The objective of some ventures “is to provide CASE tool support for patterns and to automate the use of patterns as much as possible” [8, p. 424]. Further ventures may endeavor to provide ways automate to the process of combining multiple patterns into the framework of a system. There is debate as to whether tools, such as pattern browsers, are more useful to developers than tools, such as integrated development environments or CASE tools.


RELATED REFERENCES


Douglas Schmidt, "Virtual Interview with James Coplien". C++ Report, Sept. 1995, p. 50


APPENDIX A Builder Application Source Code
Appendix A

package builder;

//Import the necessary Java packages
import java.io.*;
import java.util.*;
import java.lang.*;

/*
 * Converts token strings to an integer representation.
 * @author Stephanie Burton
 * @version 1.0, %G%
 */
class TokenType {

    // Constants for available tags for the input form
    public final static int UNDEFINED_TOKEN = 0;
    public final static int START_HEADER = 1;
    public final static int END_HEADER = 2;
    public final static int START_TITLE = 3;
    public final static int END_TITLE = 4;
    public final static int START_BODY = 5;
    public final static int END_BODY = 6;
    public final static int START_PARAGRAPH = 7;
    public final static int END_PARAGRAPH = 8;
    public final static int START_UNORDEREDLIST = 9;
    public final static int END_UNORDEREDLIST = 10;
    public final static int START_ORDEREDLIST = 11;
    public final static int END_ORDEREDLIST = 12;
    public final static int START_H1_FONT = 13;
    public final static int END_H1_FONT = 14;
    public final static int START_H2_FONT = 15;
    public final static int END_H2_FONT = 16;
    public final static int START_H3_FONT = 17;
    public final static int END_H3_FONT = 18;
    public final static int NEWLISTITEM = 19;
    public final static int NEWWORD = 20;

    // Attributes

    /*
     * The integer equivalent for the incoming token string.
     */
    private int _tagType;

    /*
     * Local copy of the incoming token string.
     */
private String _parameterStr;

/**
 * Flag to determine if incoming token string is a command.
 */
boolean _isCommand;

// Methods

/**
 * Creates a TokenType object.
 * @param tokenStr string encountered in the input file.
 * @param commandToken flag denoting whether string is a command.
 */
public TokenType(String tokenStr, boolean commandToken) {
    _tagType = UNDEFINED_TOKEN;
    _parameterStr = tokenStr.trim();
    _isCommand = commandToken;
}

/**
 * Returns the integer equivalent for the token string. An index
 * selected from the class declared constants.
 * @return integer equivalent for the token string.
 */
int getTokenType() {
    if (_isCommand) {
        if (_parameterStr.equalsIgnoreCase("head"))
            _tagType = START_HEADER;
        else if (_parameterStr.equalsIgnoreCase("/head"))
            _tagType = END_HEADER;
        else if (_parameterStr.equalsIgnoreCase("title"))
            _tagType = START_TITLE;
        else if (_parameterStr.equalsIgnoreCase("/title"))
            _tagType = END_TITLE;
        else if (_parameterStr.equalsIgnoreCase("body"))
            _tagType = START_BODY;
        else if (_parameterStr.equalsIgnoreCase("/body"))
            _tagType = END_BODY;
        else if (_parameterStr.equalsIgnoreCase("p"))
            _tagType = START_PARAGRAPH;
        else if (_parameterStr.equalsIgnoreCase("/p"))
            _tagType = END_PARAGRAPH;
        else if (_parameterStr.equalsIgnoreCase("ul"))
            _tagType = START_UNORDEREDLIST;
        else if (_parameterStr.equalsIgnoreCase("/ul"))
            _tagType = END_UNORDEREDLIST;
        else if (_parameterStr.equalsIgnoreCase("ol"))
            _tagType = START_ORDEREDLIST;
        else if (_parameterStr.equalsIgnoreCase("/ol"))
            _tagType = END_ORDEREDLIST;
        else if (_parameterStr.equalsIgnoreCase("LI"))
            _tagType = NEWLISTITEM;
    }
    return _tagType;
}
else if(_parameterStr.equalsIgnoreCase("h1"))
  _tagType = START_H1_FONT;
else if(_parameterStr.equalsIgnoreCase("/h1"))
  _tagType = END_H1_FONT;
else if(_parameterStr.equalsIgnoreCase("h2"))
  _tagType = START_H2_FONT;
else if(_parameterStr.equalsIgnoreCase("/h2"))
  _tagType = END_H2_FONT;
else if(_parameterStr.equalsIgnoreCase("h3"))
  _tagType = START_H3_FONT;
else if(_parameterStr.equalsIgnoreCase("/h3"))
  _tagType = END_H3_FONT;
else
  _tagType = UNDEFINED_TOKEN;
}
else
  _tagType = NEWWORD;

return _tagType;

/**
 * Returns the constructor's string parameter.
 */

String getParameterString()
{
  return _parameterStr;
}

// main() - This main function tests the behavior of the
// TokenTag class. For each string entered, it will print
// the entered string and its corresponding TokenTag integer
// constant.
public static void main(String[] args) throws IOException{
  String inStr,
  command;
  TokenTag tokenStr,
  BufferedReader in = new BufferedReader(new
  InputStreamReader(System.in));

  for(;;)
  {
    System.out.println("Enter a string(q to quit):");
    inStr = in.readLine();
    if(inStr.equalsIgnoreCase("q"))
      break;
    System.out.println("Is string a command (n or y)?");
    command = in.readLine();
    if((command.equalsIgnoreCase("y")) ||
      (command.equalsIgnoreCase("yes")))
      tokenStr = new TokenTag(inStr,true);
    else
      tokenStr = new TokenTag(inStr,false);

    System.out.println("Token: " +
                  tokenStr.getParameterString() + " "
                  + tokenStr.getTokenTag());
  }
A class representing the Director that parses the input file.

@author Stephanie Burton
@version %il_v_0%

public class HTMLConversionDirector {

    // Attributes

    /**
     * The concrete builder that will be directed to perform construction.
     *
     * @see ConvertedHTMLBuilder
     */
    ConvertedHTMLOutputBuilder _builder;

    /**
     * String parsed from the input file.
     *
     * @see String
     */
    String _inputItem;

    /**
     * Flag denoting current input string as a command string.
     */
    boolean _itemIsCommand;

    /**
     * Returns the character at the specified index. An index
     * ranges from <tt>0</tt> to <tt>length() - 1</tt>.
     *
     * @param index the index of the desired character.
     * @return the desired character.
     * @exception StringIndexOutOfBoundsException
     * if the index is not in the range <tt>0</tt> to <tt>length() - 1</tt>.
     */
    HTMLConversionDirector(ConverterHTMLOutputBuilder concreteBuilder) {
        _builder = concreteBuilder;
        _inputItem = "";
    }

    /**
     * Parses the input and invokes the appropriate builder method.
     *
     * @param instream the file stream to be parsed.
     * @exception IOException
     * if file stream cannot be accessed.
     */
    public void construct(Reader instream) throws IOException {

    }
}
// Set up a Tokenizer for the Input Stream and set
// the character set to recognize the right tokens
StreamTokenizer in = new StreamTokenizer(instream);
in.resetSyntax();
in.slashSlashComments(false);
in.slashStarComments(false);
in.wordChars(0, 255);
in.ordinaryChar('<');
in.ordinaryChar('>');
in.whitespaceChars(0, 36);
in.eolIsSignificant(true);

// Parse the input file separating it into the allowed
// tokens and delegate the conversion to
// the correct builder function.
while (in.nextToken() != StreamTokenizer.TT_EOF) {
    _inputItem = "";
    if (in.ttype == '<') {
        while ((in.nextToken() != StreamTokenizer.TT_EOF)
                && (in.ttype != '>')) {
            if (in.ttype == StreamTokenizer.TT_TOKEN) {
                _inputItem = _inputItem + in.sval;
            }
            _itemIsCommand = true;
        }
    } else if (in.ttype == StreamTokenizer.TT_EOL) {
        _itemIsCommand = false;
    } else
        System.out.println("Unknown Token Encountered");

    TokenType token = new TokenType(_inputItem,
                                       _itemIsCommand);

    // For each token found delegate its construction to
    // the builder.
    switch (token.getTokenType()) {
        case TokenType.UNDEFINED_TOKEN:
            break;

        case TokenType.START_HEADER:
            _builder.beginHeader();
            break;

        case TokenType.END_HEADER:
            _builder.endHeader();
            break;

        case TokenType.START_TITLE:
            _builder.beginTitle();
            break;

        case TokenType.END_TITLE:
            break;

        default:
            throw new Error("Unknown TokenType: 
                                 %s", token.getTokenType());
    }
}
case TokenType.START_BODY:
    _builder.beginBody();
    break;

case TokenType.END_BODY:
    _builder.endBody();
    break;

case TokenType.START_PARAGRAPH:
    _builder.beginParagraph();
    break;

case TokenType.END_PARAGRAPH:
    _builder.endParagraph();
    break;

case TokenType.START_UNORDEREDLIST:
    _builder.beginUnorderedList();
    break;

case TokenType.END_UNORDEREDLIST:
    _builder.endUnorderedList();
    break;

case TokenType.START_ORDEREDLIST:
    _builder.beginOrderedList();
    break;

case TokenType.END_ORDEREDLIST:
    _builder.endOrderedList();
    break;

case TokenType.START_H1_FONT:
    _builder.beginH1Font();
    break;

case TokenType.END_H1_FONT:
    _builder.endH1Font();
    break;

case TokenType.START_H2_FONT:
    _builder.beginH2Font();
    break;

case TokenType.END_H2_FONT:
    _builder.endH2Font();
    break;

case TokenType.START_H3_FONT:
    _builder.beginH3Font();
    break;

case TokenType.END_H3_FONT:
    _builder.endH3Font();
break;

case TokenType.NEWLISTITEM:
    _builder.newListItem();
    break;

case TokenType.NEWWORD:
    _builder.newWord
        (token.getParameterString());
    break;

default:
    System.out.println("Unidentified Token");
    }
}

/**
 * Starts the application. Opens the input file, creates
 * the concrete directors and builders,
 * and tells the director to start parsing.
 *
 * @param  args  command line arguments.
 * @exception IOException
 * if file can not be accessed.
 */
public static void main(String[] args) throws IOException {
    String inputFileName;
    ConvertedHTMLOutputBuilder instanceBuilder =
        new ConvertedHTMLOutputBuilder();

    // Parse the command line to determine the conversion directive
    // (toascii or tolatex) and the input file name
    if(args.length < 2){
        System.out.print("Conversion Directive and/or filename");
        System.out.println(" not specified");
        System.exit(0);
    }

    inputFileName = args[1];

    // Save the filename without the extension
    StringTokenizer nameStr = new StringTokenizer(inputFileName, ".");
    String inputFileStr = nameStr.nextToken();

    if(args[0].equalsIgnoreCase("ToAscii")){
        ConvertedHTMLOutputBuilder textBuilder =
            new Asciibuilder(inputFileStr);
        instanceBuilder = textBuilder;
    }
    else if (args[0].equalsIgnoreCase("ToLatex")){
        ConvertedHTMLOutputBuilder latexBuilder =
            new LatexWindowBuilder(inputFileStr);
        instanceBuilder = latexBuilder;
    }
else {
    System.out.println("Unknown Conversion Directive");
    System.exit(0);
}

// Create the concrete director
HTMLConversionDirector FileParser =
    new HTMLConversionDirector(instanceBuilder);

System.out.println("Converting " + inputFileName + " + args[0]);

// Open the input file, you should catch the exception
// here(error checking) if the file does not exist,
// stop here.
try{
    Reader in_stream = new BufferedReader
        (new FileReader(inputFileName));

    // Start the director's parser function
    FileParser.construct(in_stream);

    // Get the Results
    instanceBuilder.getResult();
}
catch(IOException e){
    System.out.println("Filename specified may not exist");
}
}

package builder;

/**
 * A class defining all possible actions for future builders.
 */

/* @author Stephanie Burton */
/* @version "0.1", "0.2", "0.3" */

public class ConvertedHTMLOutputBuilder{
    public void beginHeader() { }
    public void endHeader() { }
    public void beginTitle() { }
    public void endTitle() { }
    public void beginBody() { }
    public void endBody() { }
    public void beginParagraph() { }
    public void endParagraph() { }
    public void beginOrderedList() { }
}
package builder;

// Import the necessary Java packages
import java.io.*;
import java.util.Stack;
import java.util.StringTokenizer;

/**
 * A class converts incoming file information to ASCII text format.
 * @author Stephanie Burton
 * @version 1.0.0.00
 * @see ConvertedHTMLOutputBuilder
 */
public class AsciiBuilder extends ConvertedHTMLOutputBuilder {

    // Attributes
    /**
     * The maximum length of one line of text.
     */
    final static int MAXLINELENGTH = 70;

    /**
     * The name of the input file without the extension.
     */
    @see String
    String _fileStr;

    /**
     * The storage string for the converted text.
     */
    @see String
    String _product;

    // Constructor
    public AsciiBuilder () {
        // Initialization
    }
}

// File: ASCIIBuilder.java
// Purpose: Within this implementation of the Builder pattern,
// the ASCIIBuilder performs all of the construction to produce
// the converted HTML file information in ASCII format.

The counter containing the number of characters on one text line.

```
int _charOnLine;
```

```
* The flag denoting the beginning of a text line.
*
boolean _startOfLine;
```

```
* The current number for an ordered list.
*/
int _ItemCount;
```

```
* The number indents strings added to the beginning of a text line.
* An indent is represented by a string of five space characters.
*/
int _indentmultiple;
```

```
* The command string stored when the object's state is saved.
* @see String
* */
String _commandStr;
```

```
* The storage structure for saving the object's state.
* @see Stack
* */
Stack _commandStack;
```

```
//Methods
```

```
* Constructs an AsciiBuilder object.
* @param inFile input file name without extension.
*/
AsciiBuilder(String inFile) {
    _fileStr = inFile;
    _product = new String();
    _commandStr = new String();
    _charOnLine = 0;
    _indentmultiple = 0;
    _startOfLine = true;
    _commandStack = new Stack();
}
```

```
* Saves the converted output in Ascii format to a file. The
* filename is the name of the input file with the txt extension.
* @exception IOException
```
if an error occurs in creating or accessing the desired file.

```
// Save the converted Ascii Text to a file (filename.txt)
try{
    PrintWriter outfile = new PrintWriter(
        new BufferedWriter(
            new FileWriter(_fileStr + ".txt")));
    outfile.println(_product);
    outfile.flush(); // empty buffered info
    System.out.println("Output saved to " + _fileStr + ".txt");
    System.out.println("Unable to save to file");
} catch(IOException e){
    System.out.println("Unable to save to file");
}
```

```*/
* Inserts two newline characters before the title string.
*/
```

```
public void beginTitle(){
    newline();
    newline();
}
```

```*/
* Inserts two newline characters after the title string.
*/
```

```
public void endTitle(){
    newline();
    newline();
}
```

```*/
* Inserts a newline and indent before the text of the paragraph.
*/
```

```
public void beginParagraph(){
    // Start a new line
    newline();

    // Paragraph begin with an indent
    _product = _product + " "; // indent 5 spaces

    // Increase characters on line count by amount of indent
    _charOnLine = 5;
}
```

```*/
* Inserts a newline after the paragraph text.
*/
```

```
public void endParagraph(){
    newline();
}
// Inserts a word into the output. Takes into account
// the number characters on the line and adds a new line as needed.

/**
 * @param word new word string passed in from the caller.
 */
public void newWord(String word){
    if (_startOfLine){
        _product = _product + word;
        _charOnLine = _charOnLine + word.length();
        _startOfLine = false;
    } else{
        if (_charOnLine + word.length() + 1) > MAXLINELENGTH ){
            newLine();
            newWord(word);
        } else{
            _product = _product + ' ' + word;
            _charOnLine = _charOnLine + word.length() + 1;
        }
    }
}

/**
 * Performs setup for ordered list. Existing list
 * counter must be preserved.
 */
public void beginOrderedList(){
    _commandStr = "OrderedList",
    _indentmultiple = _indentmultiple + 1;
    if (_itemCount > 0){
        saveState(); //Save the counter for the last list
        newLine();
        _itemCount = 0;
    } else{
        // Initialization for a new list
        _itemCount = 0;
    }
}

/**
 * Performs clean up of completed list. Previous list
 * counters must be restored.
 */
public void endOrderedList(){
    _indentmultiple = _indentmultiple - 1;
    //restore last list counter
    restoreState();
    newLine();
}
/**
 * Performs set up for unordered list. Existing lists' 
 * state must be preserved.
 */
public void beginUnOrderedList()
{
    _commandStr = "UnOrderedList";
    _indentmultiple = _indentmultiple + 1;
    saveState(); // Save the counter for the last list
    newLine();
}

/**
 * Performs a clean up of the completed unordered list. 
 * Previous list counters must be restored and indent must be 
 * be decreased by 1.
 */
public void endUnOrderedList()
{
    _indentmultiple = _indentmultiple - 1;
    // restore last list counter
    restoreState();
    newLine();
}

/**
 * Inserts the appropriate list item identifier. A number 
 * for ordered list and a star for unordered list.
 */
public void newListItem()
{
    String enumStr = new String();
    newLine();

    if(_commandStr.equalsIgnoreCase("OrderedList")
    {
        _itemCount = _itemCount + 1;
        if (_itemCount < 10)
            enumStr = " " + _itemCount + " ";
        else
            enumStr = _itemCount + ". ";
    }
    else
        enumStr = "* ";

    _product = _product + enumStr;
    _charOnLine = _charOnLine + enumStr.length();
}

/**
 * Helper function that performs processing for new text lines.
It adds a new line character, resets counters and adds required indents.

```java
private void newline()
{
    String indent = " "; // five spaces for indent

    // move to next line
    _product = _product + '\n';

    // account for current indent
    if (_indentmultiple == 0)
        _charOnLine = 0;
    else{
        String temp = " ";
        for (int i=0; i<_indentmultiple; i++)
            temp = temp + indent;
        if (!(temp.equals(""))){
            _product = _product + temp;
            _charOnLine = _indentmultiple * indent.length();
        }
    }

    // re-initialize flag
    _startOfLine = true;
}
```

/**
 * Saves the previous list state. For ordered lists
 * stores the counter.
 */
private void saveState()
{
    String curState = _commandStr + ' ' + _itemCount;

    _commandStack.push(curState);
}

/**
 * Restores the previous list state. For ordered lists
 * restores the counter.
 */
private void restoreState()
{
    if(!_commandStack.empty()){
        String lstState = (String)_commandStack.pop();
        StringTokenizer oldState =
            new StringTokenizer(lstState);
        _commandStr = oldState.nextToken();
        Integer myInteger = new Integer(oldState.nextToken());
        _itemCount = myInteger.intValue();
    }
}

// Main Procedure to exercise the ASCII Builder class
public static void main(String[] args)
{
    String str = "ab";
    AsciiBuilder b = new AsciiBuilder("AsciiBuilderTest");
System.out.println("Executing AsciiBuilder Test");

b.beginOrderedList();
for (int i = 0; i < 15; i++) {
    b.newListitem();
    b.newWord(str);
    if (i == 5) {
        b.saveState();
        b.beginUnorderedList();
        for (int j = 0; j < 15; j++) {
            b.newListitem();
            b.newWord(str);
        }
        b.endUnorderedList();
        b.restoreState();
    }
}

b.getResult();

package builder;

// Import the necessary Java packages
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.Stack;
import java.util.StringTokenizer;

/**
 * A class representing a window on the screen.
 * @author Stephanie Burton
 * @version %I%, %G%
 * @see java.awt.Frame
 * @see java.awt.event.ActionListener
 */
class OutputWindow extends Frame implements ActionListener {

    // Attributes

    /**
     * The input file name without the extension.
     *
     * @see String
     */

    // Other code...
}
private String inFileStr;

/**
 * The converted output text according to Latex builder.
 * @see String
 */
private String convertedText;

/**
 * The text area appearing on the frame.
 * @see java.awt.TextArea
 */
private TextArea txtWindow;

/**
 * The button that saves output and exists the application.
 * @see java.awt.Button
 */
private Button bSaveClose;

/**
 * The button that exits the application without saving output.
 * @see java.awt.Button
 */
private Button bClose;

/**
 * Creates a Frame with a text area and two buttons.
 * @param inFile input file name without extension.
 */
OutputWindow(String inFile) {
    inFileStr = inFile;

    setTitle("LaTexWindowBuilder");
    Panel p = new Panel();
    bSaveClose = new Button("Save and Close");
    bClose = new Button("Close");

    bSaveClose.addActionListener(this);
    bClose.addActionListener(this);
    p.setLayout(new FlowLayout());
    p.add(bSaveClose);
    p.add(bClose);
    add("South", p);

    txtWindow = new TextArea("", 20, 80,
                            TextArea.SCROLLBARS_VERTICAL_ONLY);
    txtWindow.setEditable(false);
    add("Center", txtWindow);
}

/**

A-16
* Performs the action when a button is pressed.
* @param e an Action event received.
*/
public void actionPerformed(ActionEvent e)
{
    if(e.getSource() == bClose){
        System.out.println("Output not saved to file");
        System.exit(0);
    }
    else if(e.getSource() == bSaveClose)
    try{
        PrintWriter outfile = new PrintWriter(
            new BufferedWriter(
                new FileWriter(inFileStr + ".tex")));
        outfile.println(convertedText);
        outfile.flush(); // empty buffered info
        System.out.println("Output saved to " + inFileStr + ".tex");
    }
    catch(IOException exc){
        System.out.println("Unable to save to file");
    }
    System.exit(0);
}

/**
* Displays the output string in the text area.
* @param prod the converted text prod.
*/
public void display(String prod){
    convertedText = prod;
    txtWindow.insert(prod,0);
}

/**
* Test the operation of the output window. An index
* ranges from <tt>0</tt> to <tt>length - 1</tt>.
* @param args command line arguments.
*/
public static void main(String[] args){
    System.out.println("Executing OutputWindow Test");
    OutputWindow f = new OutputWindow("OutputWindow Test");
    f.setSize(new Dimension (600,600));
    f.display("This is the test of the Output window class");
    f.show();
}

//*************************************************************
/**
* A class that converts incoming text into Latex format.
*
public class LatexWindowBuilder extends ConvertedHTMLOutputBuilder {

    // Attributes

    /**
     * The maximum length of a line of converted text.
     */
    final static int MAXLINELENGTH = 70;

    /**
     * The storage for the converted text.
     */
    private String _product;

    /**
     * The number of characters on the current line.
     */
    private int _charOnLine;

    /**
     * The flag denoting the beginning of a new line.
     */
    private boolean _startOfLine;

    /**
     * The number of five space indents to insert at start of line.
     */
    private int _indentMultiple;

    /**
     * The flag denoting that a title string has been encountered.
     */
    private boolean _titleIsAvailable;

    /**
     * The window to display the output.
     */
    private OutputWindow _oWindow;

    // Methods

    /**
     * Creates an object to convert text to Latex format.
     */
    @param inFile input file name without extension.
    */
    LatexWindowBuilder(String inFile) {
        _product = new String();
        _charOnLine = 0;
        _startOfLine = true;
        _indentMultiple = 0;
        _titleIsAvailable = false;
_oWindow = new OutputWindow(inFile);
}

/**
 * Displays the converted text.
 */
public void getResult(){
    _oWindow.display(_product);
    _oWindow.setSize(new Dimension (600,600));
    _oWindow.show();
}

/**
 * Adds the header string to the beginning of a new line.
 */
public void beginHeader(){
    String headerStr = "\documentclass{article}";
    newline();
    _product = _product + headerStr;
    _charOnLine = _charOnLine + headerStr.length();
    _startOfLine = false;
}

/**
 * Adds the title string to the beginning of a new line.
 */
public void beginTitle(){
    String titleStr = "\title{"
    newline();
    _product = _product + titleStr;
    _charOnLine = _charOnLine + titleStr.length();
    _startOfLine = false;

    _titleIsAvailable = true;
}

/**
 * Terminates the title string.
 */
public void endTitle(){
    _product = _product + "}"
    _charOnLine = _charOnLine + 1;
}

/**
 * Adds the body string to the beginning of a new line.
 */
public void beginBody(){
    String bodyStr = "\begin{document}n;
    String makeTitleStr = "\maketitle"
    newline();
    _product = _product + bodyStr,
/**
 * Adds the ending body string to the beginning of a line.
 */
public void endBody()
{
    String endBodyStr = "\end{document}"
;

    _indentmultiple = _indentmultiple - 1;

    newLine();
    _product = _product + endBodyStr;
    _charOnLine = _charOnLine + endBodyStr.length();
    _startOfLine = false;
}

/**
 * Adds the paragraph string to the beginning of a new line.
 */
public void beginParagraph()
{
    String paragraphStr = "\paragraph{}"
;
    // Start a new line
    newLine();

    /* Paragraph begin with an indent
    _product = _product + "   "; // indent 5 spaces
    _charOnLine = 5; */

    _product = _product + paragraphStr;
    _charOnLine = _charOnLine + paragraphStr.length();

    _startOfLine = false;
}

/**
 * Adds a new word to the output string.
 */
public void newWord(String word)
{
    if (_startOfLine)
    {
        _product = _product + word;
        _charOnLine = _charOnLine + word.length();
        _startOfLine = false;
    }
    else
    {
        if (_charOnLine + word.length() + 1) >
MAXLINELENGTH) {
  newline;
  newWord(word);
} else{
  _product = _product + ' ' + word;
  _charOnLine = _charOnLine + word.length() + 1;
}
}

/**
 * Adds the ordered list string to the beginning of a new line.
 */
public void beginOrderedList() {
  String olistStr = "\begin{enumerate}"
  newline;
  _product = _product + olistStr;
  _charOnLine = _charOnLine - olistStr.length();
  _startOfLine = false;
  _indentmultiple = _indentmultiple + 1;
}

/**
 * Add the ordered list end string.
 */
public void endOrderedList() {
  String endolistStr = "\end{enumerate}"
  newline;
  _product = _product - endolistStr;
  _charOnLine = _charOnLine + endolistStr.length();
  _startOfLine = false;
}

/**
 * Inserts the string denoting unordered list to a new line.
 */
public void beginUnOrderedList() {
  String ulistStr = "\begin{itemize}"
  newline;
  _product = _product + ulistStr;
  _charOnLine = _charOnLine + ulistStr.length();
  _startOfLine = false;
  _indentmultiple = _indentmultiple - 1;
}

/**
 * Inserts the string denoting end of unordered list.
 */
public void endUnOrderedList()
{
    String endulistStr = "\end\{itemize}\";

    _indentmultiple = _indentmultiple - 1;

    newLine();
    _product = _product + endulistStr;
    _charOnLine = _charOnLine + endulistStr.length();
    _startOfLine = false;
}

/**
 * Inserts section heading string on a new line. Font type
 * H1 is treated as sections headings.
 */
public void beginH1Font()
{
    String h1Str = "\section{\";

    newLine();
    _product = _product + h1Str;
    _charOnLine = _charOnLine + h1Str.length();
    _startOfLine = false;
}

/**
 * Terminates the section heading
 */
public void endH1Font()
{
    String endh1Str = "}\";

    _product = _product + endh1Str;
    _charOnLine = _charOnLine + endh1Str.length();
}

/**
 * Inserts subsection heading string on a new line. Font type
 * H2 is treated as subsections.
 */
public void beginH2Font()
{
    String h2Str = "\subsection{\";

    newLine();
    _product = _product + h2Str;
    _charOnLine = _charOnLine + h2Str.length();
    _startOfLine = false;
}

/**
 * Inserts subsubsection heading string on a new line. Font type
 * H3 are treated as subsubsections.
 */
public void beginH3Font()
{
    String h3Str = "\subsubsection{\";

    newLine();
    _product = _product + h3Str;
    _charOnLine = _charOnLine + h3Str.length();
}
/**
* Inserts item string on a new line. Denotes the start of
* a new line item.
*/
public void newListltem() {
    String itemStr = "\item ";
    newline();
    _product = _product + itemStr;
    _charOnLine = _charOnLine + itemStr.length();
    _startOffLine = false;
}

// helper function to insert a new line and re-initialize
// character per line counter
/**
* helper function to insert a new line and re-initialize
* character per line counter and add indent if necessary.
*/
private void newline() {
    String indent = " "; // five spaces for indent
    // move to the next line
    _product = _product + '\n';
    // account for current indent
    if (_indentmultiple == 0)
        _charOnLine = 0;
    else{
        String temp="";
        for (int i=0; i<_indentmultiple; i++)
            temp = temp + indent;
        if(!(temp.equals(""))){
            _product = _product + temp;
            _charOnLine = _indentmultiple * indent.length();
        }
    }
    // re-initialize flag
    _startOffLine = true;
}

// Main Procedure to exercise the Text Window Builder class
public static void main(String[] args) {
    String str = "ab";
    LatexWindowBuilder b = new LatexWindowBuilder("LatexWindowBuilderTest");
    System.out.println("Executing LatexWindowBuilder Test");
    b.beginOrderedList();
    for (int i = 0; i < 15; i++){
        b.newListItem();
        b.newWord(str);
        if (i == 5){
            break;
        }
    }
    System.out.println("End of ordered list.");
    b.endOrderedList();
}

A-23
b.beginUnOrderedList();
for (int j = 0; j < 15; j++){
    b.newLineItem();
    b.newWord(srr);
}

b.endUnOrderedList();

b.endOrderedList();

b.getResult();
Appendix B

//*******************************************************************************
// File: DependencyTraverser.java
// Purpose: Parse the input dependency file and exercise the
// Composite Pattern structure.
//*******************************************************************************

package composite;

//Import the necessary Java packages
import java.io.*;
import java.util.*;
import java.util.StringTokenizer;

/**
 * A class that parses the input dependency file, constructs,
 * and exercises the Composite Pattern structure.
 *
 * @author Stephanie Burton
 * @version %I%, %G
 */
public class DependencyTraverser{
    // Attributes
    /**
     * The list of targets in the dependency file.
     */
    private TargetVector _targetList;

    // Methods
    /**
     * Creates a DependencyTraverser object.
     */
    public DependencyTraverser(){
        _targetList = new TargetVector();
    }

    /**
     * Parses the input dependency file.
     *
     * @param inFile the name of the input file.
     * @exception IOException if the file does not exist.
     */
    public void ParseInfile(String inFile) throws IOException {
        String line = new String();

        try{
            LineNumberReader inStream = new LineNumberReader (
            (new FileReader (inFile)));

            // Parse the input file separating it into single lines

        } catch (IOException e){
            ...} // handle exception

    }
}
/ to be parsed by a line parser
while ((line = inStream.readLine()) != null) {
    LineParser(line.trim());
}

} catch (IOException e) {
    System.out.println("Makefile.txt does not exist");
    System.exit(1);
}

/**
* Constructs the composite structure the lines in the
* input file, one line at a time.
*
* @param oneLine a single line of the input file.
*/
private void LineParser(String oneLine) {
    Target target;
    Target childTarget;
    StringTokenizer line = new StringTokenizer(oneLine, ":,;");
    int tokenCount = line.countTokens();
    if (tokenCount == 1) { // for Simple Targets
        target = new SimpleTarget(line.nextToken().trim());
        if (_targetListContentsExist(target) >= 0) {
            System.out.print("Redeclaration of target: ");
            System.out.println(target.getName());
            System.exit(1);
        }
    } else { // for Composite Target
        target = new CompositeTarget(line.nextToken().trim());
        if (_targetListContentsExist(target) >= 0) {
            System.out.print("Redeclaration of target: ");
            System.out.println(target.getName());
            System.exit(1);
        }
    }
    while (line.hasMoreTokens()) {
        int childIndex = -1;
        childTarget = new SimpleTarget(line.nextToken().trim());
        if (childIndex = _targetListContentsExist(childTarget) < 0) {
            System.out.print("Undeclared target: ");
            System.out.println(childTarget.getName());
            System.exit(1);
        } else {
            childTarget = (Target)_targetList.elementAt(childIndex);
            target.add(childTarget);
        }
    }
    _targetList.addElement(target);
/**
 * Builds all Targets specified in the dependency file.
 */
public void BuildAllTargets()
{
    for (Enumeration e = _targetList.elements(); e.hasMoreElements();)
    {
        ((Target)e.nextElement()).Build();
        System.out.println("\n");
    }
}

/**
 * Builds one of the targets specified in the dependency file.
 *
 * @param targetName the target to be built.
 */
public void BuildTarget(String targetName)
{
    Target target = new Target(targetName);
    int targetIndex = -1;

    if((targetIndex=_targetList.ContentsExist(target)) >= 0)
        ((Target)_targetList.elementAt(targetIndex)).Build();
    else
        System.out.println("Don't know how to build target: " + targetName);
}

/**
 * Lists the unique targets that each target specified in
 * the dependency file depends on.
 */
public void ViewAllTargets()
{
    for (Enumeration e = _targetList.elements(); e.hasMoreElements();)
    {
        for (Enumeration f = ((Target)e.nextElement()).View(); f.hasMoreElements();)
        {
            Target child = (Target)f.nextElement();
            if (child != null)
                System.out.println("\t" + child.GetName());
        }
        System.out.println("\n");
    }
}

/**
 * List the unique targets that the specified target depends on.
 *
 * @param targetName the target whose dependencies will be displayed.
 * @return the desired character.
 * @exception StringIndexOutOfBoundsException
 * if the index is not in the range <tt>0</tt>
 * to <tt>length()-1</tt>.
 */
public void ViewTarget(String targetName)
{
    Target target = new Target(targetName);
}
int targetIndex = -1;

if((targetIndex=_targetList.ContentsExist(target)) >= 0){
    for (Enumeration e = ((Target)_targetList.elementAt(targetIndex)).View();
        e.hasMoreElements();)
    {
        Target child = (Target)e.nextElement();
        if (child != null)
            System.out.println("\t" + child.GetName());
    }
} else
    System.out.println("Target: " + targetName + " Not found");

/**
 * The main procedure that directs the execution. The expected
 * input file is "Makefile.txt" The format requires all Simple
 * Target to be declared first, followed by declaration of Composite
 * Targets. The ';' character serves as a line terminator.
 * Example of Makefile.txt format:
 * <pre>
 *   Target1:
 *   Target2:
 *   Target3:
 *   Target5:
 *   Target6:
 *   Target4: Target1, Target2, Target3;
 *   Target7: Target5, Target6, Target4, Target1;
 *   Target8: Target4;
 *   Target9: Target8, Target7;
 * </pre>
 * @param args the command line arguments (none required).
 * @exception IOException
 *            if Makefile.txt does not exist.
 */
public static void main(String[] args) throws IOException {
    // local variables
    String inFileName = "Makefile.txt";

    // Create an instance of DepdencyTraverser
    DependencyTraverser deptrav = new DependencyTraverser();

    // Parse the input file to build the Composite Structure
    deptrav.ParseInfile(inFileName);

    System.out.println("Building All Targets:");
    deptrav.BuildAllTargets();

    //System.out.println("Building Target9");
    //deptrav.BuildTarget("Target9");

    System.out.println("\nViewing all Targets");
    deptrav.ViewAllTargets();

    //System.out.println("\nViewing Target7");
    //deptrav.ViewTarget("Target7");
package composite;

//Import the necessary Java packages
import java.util.*;

public class TargetVector extends Vector{
    // Attributes

    // Methods
    public TargetVector(){ }

    // test if the contents of a test Target exists in an element
    // that is already in the vector.
    public int ContentsExist(Target testTarget) {
        for (Enumeration e = elements(); e.hasMoreElements();)
            Target element = (Target)e.nextElement();
            if (element.getName().equals(testTarget.getName()))
                return indexOf(element);
        
        return -1;
    }
}

package composite;

import java.util.*;

/**
 * A concrete class representing the common interface for all Targets.
 * 
 * @author Stephanie Burton
 * @version %I%, %G%
 */
public class Target{
    // Attributes

        /**
        * The name of the Target.
        */
}
String _name;

// Methods

/**
 * Creates a Target with the specified name.
 * @param nameStr the name of the Target.
 */
public Target(String nameStr)
{
    _name = nameStr;
}

/**
 * Returns the name of the Target.
 * @return the name of the Target.
 */
public String GetName()
{
    return _name;
}

/**
 * Generates a list of all unique Targets that this Target depends on.
 * @return the Enumeration for the list.
 */
public Enumeration View()
{
    // This is a blank vector used to return a dummy enumeration
    Vector temp = new Vector();
    return temp.elements();
}

/**
 * Performs functions to build the Target
 */
public void Build()
{
}

/**
 * Adds the specified Target to the list of dependent Targets.
 * @param depTarget Target that this Target depends on.
 */
public void Add(Target depTarget)
{
}

// Package: composite

// Include necessary class files
import java.io.*;
import java.util.*;

/**
 * A class representing a Target that may have dependencies on
 * other targets (Simple or Composite).
 * @author Stephanie Burton
 * @version %I%, %G%
 */
public class CompositeTarget extends Target{
    // Attributes

    /**
     * List of Targets that this CompositeTarget depends on.
     */
    private TargetVector dependencyList;

    /**
     * List of unique Targets that this CompositeTarget depends on.
     */
    private TargetVector viewList;

    // Methods

    /**
     * Creates a Composite Target with the supplied
     * name string.
     *
     * @param nameStr the name of the Composite Target.
     */
    public CompositeTarget(String nameStr){
        super(nameStr);
        dependencyList = new TargetVector();
        viewList = new TargetVector();
    }

    /**
     * Adds the supplied target to the Composite Targets
     * list of dependencies.
     *
     * @param depTarget a target that the CompositeTarget depends on.
     */
    public void Add(Target depTarget){
        dependencyList.addElement(depTarget);
    }

    /**
     * Constructs a vector of unique Targets
     * that the Composite Target Depends on.
     *
     * @return Enumeration of the constructed vector.
     */
    public Enumeration View(){
        TargetVector childList = new TargetVector();
        for (Enumeration e = dependencyList.elements(); e.hasMoreElements();)
            Target child = (Target)e.nextElement();

        return childList;
    }
}
if (child != null) {
    for (Enumeration f = child.View();
         f.hasMoreElements();)
    {
        Target dependent = (Target)f.nextElement();
        if (dependent != null) {
            if((viewList.isEmpty()) ||
               (viewList.ContentsExist(dependent) < 0))
                viewList.addElement(dependent);
        }
    }
}

if(viewList.ContentsExist(this) < 0)
    viewList.addElement(this);
return viewList.elements();

/**
 * Builds the Composite Target after building all targets
 * that the target depends on.
 */
*/
public void Build() {
    System.out.println("BUILDING COMPOSITE TARGET: " + GetName());
    for (Enumeration e = dependencyList.elements(); e.hasMoreElements();)
    {
        Target child = (Target)e.nextElement();
        if (child != null) {
            child.Build();
        }
    }
    System.out.println("COMPLETED COMPOSITE TARGET: ");
    System.out.println(GetName());
}

/**
 * Test program to exercise the functionality of
 * Composite Target class.
 */
*/
public static void main(String[] args) {
    CompositeTarget target3 = new CompositeTarget("target3");
    Target target2 = new SimpleTarget("target2");
    Target target1 = new SimpleTarget("target1");
    Target target0 = new SimpleTarget("target0");
    target3.Add(target2);
    target3.Add(target1);
    target3.Add(target0);
    CompositeTarget target7 = new CompositeTarget("target7");
}
Target target6 = new SimpleTarget("target6");
Target target5 = new SimpleTarget("target5");
Target target4 = new SimpleTarget("target4");

target7.Add(target1);
target7.Add(target3);
target7.Add(target4);
target7.Add(target5);
target7.Add(target3);
target7.Add(target6);

System.out.println("Building Targets:");
target7.Build();

System.out.println("Viewing Targets:");
for (Enumeration e = target7.View(); e.hasMoreElements();)
    {
        Target child = (Target)e.nextElement();
        if (child != null)
            System.out.println("\t" + child.GetName());
    }
System.out.println(" ");

package composite;

// Include necessary class files
import java.io.*;
import java.util.*;

/**
 * A class representing a target that does not depend on other targets.
 *
 * @author Stephanie Burton
 * @version %I%, %G%
 */
public class SimpleTarget extends Target{
    // Attributes

    /**
     * Vector containing the single target as its only dependent
     */
    private TargetVector viewList;

    // Methods

    /**
     * Creates a target with the specified name.
     *
     * @param nameStr the name of the target.
     */
    public SimpleTarget(String nameStr){

    }
super(nameStr);

/**
 * Adds itself to its dependency vector and
 * Returns the Enumeration for the constructed vector.
 */
public Enumeration View(){
    viewList = new TargetVector();

    viewList.addElement(this);

    return viewList.elements();
}

/**
 * Prints out message denoting the end of its build.
 */
public void Build(){
    System.out.print("Completed Building Simple Target: ");
    System.out.println(GetName());
}

/**
 * Test procedure to exercise the SimpleTarget class methods.
 */
public static void main(String[] args){
    SimpleTarget target1 = new SimpleTarget("target1");

    for (Enumeration e = target1.View(); e.hasMoreElements(); ) {
        System.out.println(((Target)e.nextElement()).GetName());
    }

    target1.Build();
}
APPENDIX C Adapter Application Source Code
Appendix C

//*******************************************************************************************
// File: ButtonListener.java
// Purpose: Creates a 1.1 panel with a 1.0 image button
//*******************************************************************************************

public class ButtonListener implements ActionListener {

    public ButtonListener() {

    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("1.0 Button was pressed so exit");
        System.exit(0);
    }
}

//*******************************************************************************************
// File: ImageButton.java
// Purpose: A button class that uses an image instead of a textual label
//*******************************************************************************************

public class ImageButton {

    // For ImageFilter stuff
    // TO DO: Make button resizable and stretch the image to fit.
}
A button class that uses an image instead of a textual label. Unlike many of the ImageButton classes available, it triggers an ACTION_EVENT, so you can add behavior in the same two ways as you can with normal Buttons:

- Make a subclass and put the behavior in the subclass's `action` method.
- Use the main class but then catch the event in the Container's `action` method.

By default, with FlowLayout the ImageButton takes its minimum size (just enclosing the image). The default with BorderLayout is to expand to fill the region in width (North/South), height (East/West) or both (Center). This is the same behavior as the builtin Button class. If you give an explicit `resize` or `reshape` call before adding the ImageButton to the Container, this size will override the defaults.

The original of the source code can be found at http://www.apl.jhu.edu/~hall/java/ImageLabel/ImageButton.java, the documentation is at http://www.apl.jhu.edu/~hall/java/ImageLabel/ImageButton.html, and a small example program can be found at http://www.apl.jhu.edu/~hall/java/ImageLabel/ImageButtonTest.html.

No warranty of any kind is provided. Permission is granted to use and/or modify for any purpose.

---

```java
public class ImageButton extends ImageLabel {
    // Default width of 3D border around image. Currently 4.
    protected static final int defaultBorderWidth = 4;
    // Default color of 3D border around image. Currently a gray
```

protected static final Color defaultBorderColor = new Color(160, 160, 160);

// Darker is consistent with regular buttons
/** An int whose bits are combined via "and" ("&") with the alpha, 
* red, green, and blue bits of the pixels of the image to produce 
* the grayed-out image to use when button is depressed. 
* Default is 0xffafafaf: af combines with r/g/b to darken image. 
*/
public int darkness = 0xffafafaf;

private boolean mouseIsDown = false;

// Constructor

/** Create an ImageButton with the default image. 
* @see ImageLabel#defaultImageString 
*/
public ImageButton() {
    super();
    setDefaults();
}

/** Create an ImageButton using the image at URL specified by the string. 
* @param imageURLString A String specifying the URL of the image. 
*/
public ImageButton(String imageURLString) {
    super(imageURLString);
    setDefaults();
}

/** Create an ImageButton using the image at URL specified. 
* @param imageURL The URL of the image. 
*/
public ImageButton(URL imageURL) {
    super(imageURL);
    setDefaults();
}

/** Create an ImageButton using the image specified. You would only want to 
* call it directly if you already have an image (e.g. created via 
* createImage). 
* @param image The image. 
*/
public ImageButton(Image image) {
    super(image);
    setDefaults();
}
/** Sets the width around the outside of the image. */
@see #getBorderWidth

public void setBorderWidth(int borderWidth) {
    margin = borderWidth;
}

/** Returns the width around the outside of the image. */
@see #setBorderWidth

public int getBorderWidth() {
    return (margin);
}

/** Sets the Color for the border around the outside of the image. */
@see #getBorderColor

public void setBorderColor(Color borderColor) {
    marginColor = borderColor;
}

/** Returns the Color for the border around the outside of the image. */
@see #setBorderColor

public Color getBorderColor() {
    return (marginColor);
}

/** Draws the image with the border around it. If you override this in a subclass, call super.paint(). */

public void paint(Graphics g) {
    super.paint(g);  // Forces main image to get loaded
    if (grayImage == null)  // Creates gray image from main image
        createGrayImage(g);
    drawBorder(true);
}

/** When the mouse is clicked, reverse the 3D border and draw a dark-gray version of the image. The action is not triggered until mouseUp. */

public boolean mouseDown(Event event, int x, int y) {
    mouseIsDown = true;
    Graphics g = getGraphics();
    if (explicitSize)
g.drawImage(grayImage, margin, margin,
   width-2*margin, height-2*margin,
   this);
else
   g.drawImage(grayImage, margin, margin, this);
drawBorder(false);
return(true);
}

// Generated when the button is clicked and released. Override this
/**
 * in subclasses to give behavior to the button. Alternatively, since
 * the default behavior is to pass the ACTION_EVENT along to the
 * Container, you can catch events for a bunch of buttons there.
 * @see Component#action
 */
public boolean mouseUp(Event event, int x, int y) {
   mouseIsDown = false;
   if (inside(x,y)) {
      paint(getGraphics());
      event.id = Event.ACTION_EVENT;
      event.arg = (Object)image;
      return(action(event, event.arg));
   } else
      return(false);
}

// If you move the mouse off the button while the mouse is down,
/**
 * abort and do NOT trigger the action. Ignore this if
 * button was not already down.
 */
public boolean mouseExit(Event event, int x, int y) {
   if (mouseIsDown)
      paint(getGraphics());
   return(true);
}

//

private void drawBorder(boolean isUp) {
   Graphics g = getGraphics();
   g.setColor(marginColor);
   int left = 0;
   int top = 0;
   int width = this.width;
int height = this.height;
for(int i=0; i<margin; i++) {
    g.draw3DRect(left, top, width, height, isUp);
    left++;
    top++;
    width = width - 2;
    height = height - 2;
}

private void setDefaults() {
    margin = defaultBorderWidth;
    marginColor = defaultBorderColor;
}

// The first time the image is drawn, update() is called, and
// the printout does not come out correctly. So this forces a
// brief draw on loadup, replaced by real, non-gray image.

private void createGrayImage(Graphics g) {
    ImageFilter filter = new GrayFilter(darkness);
    ImageProducer producer = new FilteredImageSource(image.getSource(), filter);
    grayImage = createImage(producer);
    if (explicitSize)
        g.drawImage(grayImage, margin, margin, width-2*margin, height-2*margin, this);
    else
        g.drawImage(grayImage, margin, margin, this);
    super.paint(g);
}

/**
 * Builds an image filter that can be used to gray-out the image.
 * @see ImageButton
 */
class GrayFilter extends RGBImageFilter {

    private int darkness = 0xff808080;

    public GrayFilter() {
        canFilterIndexColorModel = true;
    }
    public GrayFilter(int darkness) {
this();
this.darkness = darkness;
}

//************************************************************************

public int filterRGB(int x, int y, int rgb) {
    return(rgb & darkness);
}

//************************************************************************

package adapter;

import java.awt.*;
import java.net.*;

// ************************************************************************

/**
 * A class for displaying images. It places the Image
 * into a canvas so that it can moved around by layout managers,
 * will get repainted automatically, etc. No <TT>mouseXXX</TT> or
 * <TT>action</TT> events are defined, so it is most similar to the
 * <TT>Label</TT> Component.
 * *
 * By default, with FlowLayout the ImageLabel takes its minimum size
 * (just enclosing the image). The default with BorderLayout is
 * to expand to fill the region in width (North/South), height
 * (East/West) or both (Center). This is the same behavior as the builtin
 * Label class. If you give an explicit <TT>resize</TT> or
 * <TT>reshape</TT> call <B>before</B> adding the ImageLabel to the
 * Container, this size will override the defaults.
 * *
 * Here is an example of its use:
 * *
 * public class ShowImages {
 *     ImageLabel duke, javaMug;
 *     *
 *     public void init() {
 *         add(duke);
 *         add(javaMug);
 *     }
 * }
 * }
 * *
 * The latest version of the source code is
 * *
 * <A href="http://www.apl.jhu.edu/~hall/java/ImageLabel/ImageLabel.java">
public class ImageLabel extends Canvas {
  // Instance variables.

  /** The actual Image drawn on the canvas. Don't specify this directly; */
  protected Image image;

  /** A String corresponding to the URL of the image you will get */
  public static String defaultImageString = "http://java.sun.com/lib/images/logo.java.color-transp.55x60.gif";

  /** The URL of the image. But sometimes we will use an existing image */
  protected String imageString = "<Existing Image>";

  /** Turn this on to get verbose debugging messages. */
  public boolean debug = false;

  /** Amount of extra space around the image. */
  protected int margin = 0;

  /** If there is a non-zero margin, what color should it be? Default */
  protected Color marginColor = null;
}
/** Width and height of the Canvas. This is the width/height of the image * plus twice the margin. */
protected int width, height;

/** Determines if it will be sized automatically. * If the user issues a resize() or reshape() call before adding the * label to the Container, or if the LayoutManager resizes before * drawing (as with BorderLayout), then those sizes override * the default, which is to make the label the same size as the image * it holds (after reserving space for the margin, if any). * This flag notes this, so subclasses that override ImageLabel * need to check this flag, and if it is true, and they * draw modified image, then they need to draw them based on the width * height variables, not just blindly drawing them full size. */
protected boolean explicitSize = false;
private int explicitWidth = 0, explicitHeight = 0;

// The MediaTracker that can tell if image has been loaded // before trying to paint it or resize based on its size.
private MediaTracker tracker;

// Used by MediaTracker to be sure image is loaded before paint & resize, // since you can't find out the size until it is done loading.
private static int lastTrackerID = 0;
private int currentTrackerID;
private boolean doneLoading = false;

private Container parentContainer;

/** Create an ImageLabel with the default image. * @see ImageLabel#defaultImageString */
// Remember that the funny "thisO" syntax calls constructor of same class
public ImageLabel() {
  this(defaultImageString);
}

/** Create an ImageLabel using the image at URL specified by the string. * @param urlString A String specifying the URL of the image. */
public ImageLabel(String urlString) {
  this(getImage(urlString));
}

/** Create an ImageLabel using the image at URL specified. * @param urlString The URL of the image. */
public ImageLabel(URL urlString) {
  this(getImage(urlString));
  urlString = urlString.toExternalForm();
}
/** Create an ImageLabel using the image specified. The other * constructors eventually call this one, but you may want to call * it directly if you already have an image (e.g. created via * createImage). * @param image The image */

public ImageLabel(Image image) {
    this.image = image;
    tracker = new MediaTracker(this);
    currentTrackerID = lastTrackerID++;
    tracker.addImage(image, currentTrackerID);
}

/**
 * Makes sure that the Image associated with the Canvas is done loading * before returning.
 * getImage spins off a separate thread to do the loading. Once you * get around to drawing the image, this will make sure it is loaded, * waiting if not. The user does not need to call this at all, but * if several ImageLabels are used on the same Container, this can cause * several repeated layouts, so users might want to explicitly call this * themselves before adding the ImageLabel to the Container. On the * other hand, postponing the waiting as long as possible is more * efficient, since loading can go on in the background.
 *
 * @param doLayout Determines if the Container should be re-leyed out * after you are finished waiting. <B>This should be * true when called from user functions</B>, but * is set to false when called from preferredSize * to avoid an infinite loop. This is needed when * using BorderLayout, which calls preferredSize * <B>before</B> calling paint.
 */

public void waitForImage(boolean doLayout) {
    if (!doneLoading) {
        debug("[waitForImage] - Resizing and waiting for " + imageString);
        try {
            tracker.waitForID(currenfTrackerID);
        } catch (InterruptedException i) {} catch (Exception e) {
            System.out.println("Error loading " + imageString + ": ";
            e.printStackTraceO;
            e.printStackTraceO;
        }
    }
    if (tracker.isErrorID(0))
        new Throwable("Error loading image " + imageString).printStackTraceO;
    doneLoading = true;
    if (explicitWidth != 0)
        width = explicitWidth;
    else
        width = image.getWidth(this) + 2*margin;
    if (explicitHeight != 0)
        height = explicitHeight;
    else
        height = image.getHeight(this) + 2*margin;
    resize(width, height);
    debug("[waitForImage] - " + imageString + " is " + 
        width + "x" + height + ");
}
If no parent, you are OK, since it will have been resized before being added. But if parent exists, you have already been added, and the change in size requires re-layout.

```java
if (parentContainer = getParent() != null && doLayout) {
    setBackground(parentContainer.getBackground());
    parentContainer.layout();
}
}
```

/**
 * Moves the image so that it is centered at the specified location,
 * as opposed to the move method of Component which places the top left
 * corner at the specified location.
 *
 * @param x The X coord of center of the image
 *          (in parent's coordinate system)
 * @param y The Y coord of center of the image
 *          (in parent's coordinate system)
 * @see java.awt.Component#move
 */

public void centerAt(int x, int y) {
    debugfPlacing"center of " + imageString + " at (" + x + "," + y + ");
    move(x - width/2, y - height/2);
}

/**
 * Determines if the x and y <B>(in the Icon's own coordinate
 * system)</B> is inside the Icon. Put here because
 * Netscape has a bug in which it doesn't process inside() and locate()
 * tests correctly. They work properly with appletviewer and
 * for applications.
 */

public synchronized boolean inside(int x, int y) {
    return (x >= 0) && (x <= width) && (y >= 0) && (y <= height);
}

// NOTE: drawRect on Solaris 2.4, under Netscape 2.0x, appletviewer
// from JDK 1.02, and in standalone apps under 1.02, has a bug in that
// it draws 1 pixel too far. fillRect and draw3DRect do not suffer from
// this problem, and I have not yet confirmed if it exists on
// Windows or MacOS. If the problem does *not* exist on those platforms,
// then the width-1 and height-1 below should be replaced with simply
// width and height.
/** Draws the image. If you override this in a subclass, be sure
* to call super.paint.
*/
public void paint(Graphics g) {
  if (!doneLoading)
    waitForImage(true);
  else {
    if (explicitSize)
      g.drawImage(image, margin, margin,
                  width-2*margin, height-2*margin,
                  this);
    else
      g.drawImage(image, margin, margin, this);
    drawRect(g, 0, 0, width-1, height-1, margin, marginColor);
  }
}

/**
* Used by layout managers to calculate the usual size allocated for
* the Component. Since some layout managers (e.g. BorderLayout) may
* call this before paint is called, you need to make sure
* that the image is done loading, which will force a resize, which
* determines the values returned.
*/
public Dimension preferredSize() {
  if (!doneLoading)
    waitForImage(false);
  return(super.preferredSize());
}

/**
* Used by layout managers to calculate the smallest size allocated for the
* Component. Since some layout managers (e.g. BorderLayout) may
* call this before paint is called, you need to make sure
* that the image is done loading, which will force a resize, which
* determines the values returned.
*/
public Dimension minimumSize() {
  if (!doneLoading)
    waitForImage(false);
  return(super.minimumSize());
}

/**
// LayoutManagers (such as BorderLayout) might call resize or reshape
// with only 1 component of width/height non-zero. In such a case, //
// you still want the other component to come from the image itself.

/**
* Resizes the ImageLabel. If you don't resize the label explicitly,
* then what happens depends on the layout manager. With FlowLayout,
* as with FlowLayout for Labels, the ImageLabel takes its minimum
* size, just enclosing the image. With BorderLayout, as with
* BorderLayout for Labels, the ImageLabel is expanded to fill the
* section. Stretching GIF files does not always result in clear looking
* images. <B>So just as with builtin Labels and Buttons, don't
* use BorderLayout if you don't want the Buttons to get resized.</B>
* If you don't use any LayoutManager, then the ImageLabel will
*/
just fit the image.

Note that if you resize explicitly, you must do it before the ImageLabel is added to the Container. In such a case, the explicit size overrides.

@see #reshape

/** Reshapes the ImageLabel. If you don't resize the label explicitly, then what happens depends on the layout manager. With FlowLayout, as with FlowLayout for Labels, the ImageLabel takes its minimum size, just enclosing the image. With BorderLayout, as with BorderLayout for Labels, the ImageLabel is expanded to fill the section. Stretching GIF files does not always result in clear looking images. So just as with builtin Labels and Buttons, don't use BorderLayout if you don't want the Buttons to get resized. If you don't use any LayoutManager, then the ImageLabel will just fit the image.

Note that if you resize/reshape explicitly, you must do it before the ImageLabel is added to the Container. In such a case, the explicit size overrides.

@see #resize
*/

public void reshape(int x, int y, int width, int height) {
    if (!doneLoading) {
        expiresIn=true;
        if (width > 0)
            expiresInWidth=width;
        if (height > 0)
            expiresInHeight=height;
    }
    super.reshape(x, y, width, height);
}

/** Draws a rectangle with the specified OUTSIDE left, top, width, and height. Used to draw the border. */
protected void drawRect(Graphics g, int left, int top,
             int width, int height, int lineThickness,
Color rectangleColor) {
    g.setColor(rectangleColor);
    for(int i=0; i<lineThickness; i++) {
        g.drawRect(left, top, width, height);
        if (i < lineThickness-1) { // Skip these the last rectangle
            left = left + 1;
            top = top + 1;
            width = width - 2;
            height = height - 2;
        }
    }
}

//-------------------------------------------------------------------------------
/** Calls System.out.println if the debug variable is true,
* does nothing otherwise.
* @param message The String to be printed.
*/
protected void debug(String message) {
    if (debug)
        System.out.println(message);
}

//-------------------------------------------------------------------------------
private static URL makeURL(String s) {
    URL u = null;
    try { u = new URL(s); }
    catch (MalformedURLException m) {
        System.out.println("Bad URL "+ s + ": " + m);
        m.printStackTrace();
    }
    return(u);
}

//-------------------------------------------------------------------------------
// Needs to be static since it is called by the constructor.

private static Image getImage(URL url) {
    return(Toolkit.getDefaultToolkit().getImage(url));
}

//-------------------------------------------------------------------------------
private static Image getImage(String filename) {
    return(Toolkit.getDefaultToolkit().getImage(filename));
}


//-------------------------------------------------------------------------------
// File: ImageButtonAdapter.java
// Purpose: Creates a 1.1 panel with a 1.0 image button
//-------------------------------------------------------------------------------

package adapter;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;

/**
 * A class representing a Java 1.0 button in the
 * center of a Java 1.1 panel. When pressed 1.0 button generates
 * 1.0 events. The 1.1 panel adapts the 1.0 events and
 * handles them in a 1.1 manner with the help of any
 * registered listeners (which a part of 1.1 protocol).
 * 
 * @author Stephanie Burton
 * @version %I%, %G%
 */

public class ImageButtonAdapter extends Panel{
    ImageButton button10;
    String imageLabel;
    String defaultImageLabel = "orb.gif";
    String actionCommand;
    ActionListener actionListener = null;

    /**
     * Creates an ImageButtonAdapter with the specified image as
     * the button label.
     *
     * @param  buttonImage  the string name of the image file.
     */
    public ImageButtonAdapter(String buttonImage){
        imageLabel = buttonImage;

        setLayout(new FlowLayout());

        button10 = new ImageButton(imageLabel);
        button10.debug = false;

        add(button10);
    }

    /**
     * Creates an ImageButtonAdapter with a default image as
     * the button label.
     *
     * @param  buttonImage  the string name of the image file.
     */
    public ImageButtonAdapter(){
        imageLabel = defaultImageLabel;

        setLayout(new FlowLayout());

        button10 = new ImageButton(imageLabel);
        button10.debug = false;

        add(button10);
    }
}
1.1 Button Interface

/**
 * Returns the name image file of the button image.
 * @see #setLabel
 */
public String getLabel(){
    return imageLabel;
}

/**
 * Adds the specified action listener to receive action events
 * from this button.
 * @param l the action listener
 */
public void addActionListener(ActionListener l){
    actionListener = AWTEventMulticaster.add(actionListener, l);
    // newEventsOnly = true;

    // for single listener
    //mySingleListener = l;
}

/**
 * Removes the specified action listener so it no longer receives
 * action events from this button.
 * @param l the action listener
 */
public void removeActionListener(ActionListener l){
    actionListener = AWTEventMulticaster.remove(actionListener, l);

    //for single listener
    //mySingleListener = null;
}

/**
 * Processes events on this button. If the event is an ActionEvent,
 * it invokes the processActionEvent method, else it invokes its
 * superclass's processEvent.
 * @param e the event
 */
protected void processEvent(AWTEvent e){
    if (e instanceof ActionEvent) {
        processActionEvent((ActionEvent)e);
        return;
    }
    super.processEvent(e);
}

/**
 * Processes action events occurring on this button by
 * dispatching them to any registered ActionListener objects.
 * NOTE: This method will not be called unless action events
are enabled for this component; this happens when one of the
following occurs:
* a) An ActionListener object is registered via addActionListener()
* b) Action events are enabled via enableEvents()
* @see Component#enableEvents
* @param e the action event
*/
protected void processActionEvent(ActionEvent e) {
    if (actionListener != null) {
        actionListener.actionPerformed(e);
    }
}

// ************************************************************************
// 1.0 ImageButton interface
/**
* Creates an ImageButtonAdapter with the specified image as
* the button label.
* @param evt the 1.0 event generated by the button.
* @param arg the source object of the event.
* @return boolean noting successful handling of the event.
*/
public boolean action(Event evt, Object arg) {
    ActionEvent aevt = new ActionEvent(this, 10, "ButtonPress");
    processActionEvent(aevt);
    return (true);
}
/**
* Allows the client to set the size of the border around the image
* button.
* @param borderWidth the desired size of the button border.
*/
public void setBorderWidth(int borderWidth) {
    button10.setBorderWidth(borderWidth);
}

// ************************************************************************
// main function
/**
* Exercises the ImageButtonAdapter Functionality.
* @param args the command line arguments (none required).
*/
public static void main(String[] args) {
    Frame myframe = new Frame();
    myframe.setLayout(new FlowLayout());

    ImageButtonAdapter myadapter = new 
    ImageButtonAdapter("School.gif");
    myframe.add("Center", myadapter);

    ButtonListener myListener = new ButtonListener();
myadapter.addActionListener(myListener);
myadapter.removeActionListener(myListener);
myadapter.addActionListener(myListener);
myframe.setSize(new Dimension (600,600));
myframe.show();
System.out.println(myadapter.getLabel());
myadapter.setBorderWidth(15);
APPENDIX D Mediator Application Source Code
Appendix D

import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

/**
 * A class that creates an applet and manages the interaction
 * between the components.
 *
 * @author Stephanie Burton
 */
public class MediatorApp extends Applet implements ItemListener,
     AdjustmentListener, ActionListener{

    /* Panel that holds all components in the applet.
     */
    private Panel basePanel;

    /* Panel to display the text string.
     */
    private Panel textPanel;

    /* Text string to be displayed in chosen font styles.
     */
    private Label textLabel;

    /* Panel that contains the center elements of the applet in a grid format.
     */
    private Panel gridPanel;

    /* Panel that contains font text field.
     */
    private Panel font1Panel;

    /* Text field that displays the chosen font string (not editable).
     */
    private Label font1Label;
private TextField fontField;

/**
 * Panel that contains the Label for the font field.
 */
private Panel font2Panel;

/**
 * Label for the font text field.
 */
private Label fontFieldLabel;

/**
 * Panel to contain the list of font names.
 */
private Panel listPanel;

/**
 * List of font names.
 */
private List fontList;

/**
 * Panel to contain Label for the font weight selections.
 */
private Panel weight1Panel;

/**
 * Label for the font weight selections.
 */
private Label weightgrpLabel;

/**
 * Panel to contain the font weight selection buttons.
 */
private Panel weight2Panel;

/**
 * Variable denoting the group of weight selection check boxes.
 */
private CheckboxGroup weightgrp;

/**
 * Checkbox for the "plain" font weight selection.
 */
private Checkbox plainChkbox;
* Checkbox for the "bold" font weight selection.
* 
private Checkbox boldChkbox;

/**
* Panel that contains the label for the font style selections.
* 
*/
private Panel slant1Panel;

/**
* Label for the font style selections.
* 
*/
private Label slantgrpLabel;

/**
* Panel that contains the font style selections.
* 
*/
private Panel slant2Panel;

/**
* Variable denoting the group of font style selection checkboxes.
* 
*/
private CheckboxGroup slantgrp;

/**
* Checkbox for the "no slant" font style selection.
* 
*/
private Checkbox noslantChkbox;

/**
* Checkbox for the "italic" font style selection.
* 
*/
private Checkbox italicChkbox;

/**
* Panel that contains the font size label.
* 
*/
private Panel size1Panel;

/**
* Label for the font size text field.
* 
*/
private Label sizeFieldLabel;

/**
* Panel that contains the font size text field.
* 
*/
private Panel size2Panel;
  /**
   * Text field that displays the font size (not editable).
   *
   */
private TextField sizeField;
  /**
   * Scrollbar used to selected the font sizes.
   *
   */
private Scrollbar sizeBar;
  /**
   * Panel containing the apply and exit buttons.
   *
   */
private Panel buttonPanel;
  /**
   * Button that initiates the text string change based on the current selections.
   *
   */
private Button applyButton;
  /**
   * Button that terminates the applet.
   *
   */
private Button exitButton;
  /**
   * Creates the components in the applet and arranges them
   * in the applet window. This method is required by derived
   * applet classes. It is called by the system to initialize
   * the applet.
   *
   */
public void init()
  {
    //Change applets layout manager
    //setLayout(new BorderLayout());
    //Create objects
    textLabel = new Label("The quick brown fox...", Label.CENTER);
    fontFieldLabel = new Label("Family");
    fontField = new TextField(" ", 15);
    fontField.setEditable(false);
    fontList = new List(3, false);
    fontList.addItem("Courier");
    fontList.addItem("Helvetica");
    fontList.addItem("Times Roman");
  }
fontList.addItemListener(this);

weightgrpLabel = new Label("Weight");
weightgrp = new CheckboxGroup();
plainChkbox = new Checkbox("plain",weightgrp, true);
boldChkbox = new Checkbox("bold",weightgrp,false);

slantgrpLabel = new Label("Slant");
slantgrp = new CheckboxGroup();
noslantChkbox = new Checkbox("plain",slantgrp, true);
itachChkbox = new Checkbox("italic",slantgrp, false);

sizeFieldLabel = new Label("Size (point)");
sizeField = new TextField(" ", 3);
sizeField.setEditable(false);
sizeBar = new Scrollbar(Scrollbar.VERTICAL, 0, 0, 1, 5);
sizeBar.addAdjustmentListener(this);

applyButton = new Button("Apply");
applyButton.addActionListener(this);
exitButton = new Button("Exit");
exitButton.addActionListener(this);

//Add objects to Layout

basePanel = new Panel(new BorderLayout());
textPanel = new Panel();
textPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
textPanel.add(textLabel);
basePanel.add(textPanel, "North");

gridPanel = new Panel();
gridPanel.setLayout(new GridLayout(5,2,0,-5));
font1Panel = new Panel();
font1Panel.setLayout(new FlowLayout(FlowLayout.RIGHT));
font1Panel.add(fontFieldLabel);
gridPanel.add(font1Panel);

font2Panel = new Panel();
font2Panel.setLayout(new FlowLayout(FlowLayout.LEFT));
font2Panel.add(fontField);
gridPanel.add(font2Panel);

gridPanel.add(new Label(" ") ); // space holder
listPanel = new Panel();
listPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
listPanel.add(fontList);
gridPanel.add(listPanel);

weight1Panel = new Panel();
weight1Panel.setLayout(new FlowLayout(FlowLayout.RIGHT));
weight1Panel.add(weightgrpLabel);
gridPanel.add(weight1Panel);
weight2Panel = new Panel();

D-5
weight2Panel.setLayout(new FlowLayout(FlowLayout.LEFT));
weight2Panel.add(plainChkbox);
weight2Panel.add(boldChkbox);
gridPanel.add(weight2Panel);

slant1Panel = new Panel();
slant1Panel.setLayout(new FlowLayout(FlowLayout.RIGHT));
slant1Panel.add(slantgrpLabel);
gridPanel.add(slant1Panel);
slant2Panel = new Panel();
slant2Panel.setLayout(new FlowLayout(FlowLayout.LEFT));
slant2Panel.add(noslantChkbox);
slant2Panel.add(italicChkbox);
gridPanel.add(slant2Panel);

size1Panel = new Panel();
size1Panel.setLayout(new FlowLayout(FlowLayout.RIGHT));
size1Panel.add(sizeFieldLabel);
gridPanel.add(size1Panel);
size2Panel = new Panel();
size2Panel.setLayout(new FlowLayout(FlowLayout.LEFT));
size2Panel.add(sizeField);
size2Panel.add(sizeBar);
gridPanel.add(size2Panel);

basePanel.add(gridPanel, "West");

buttonPanel = new Panel();
buttonPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
buttonPanel.add(applyButton);
buttonPanel.add(exitButton);
basePanel.add(buttonPanel, "South");

add(basePanel);

setDefaults();

resize(650,450);

repaint();

/**
 * Sets the default selections for the applets initial display.
 *
*/

void setDefaults(){
    fontList.select(0);
    fontField.setText(fontList.getSelectedItem());
    sizeField.setText("48");

    apply();
}

/**
 * Allows the applet to listen to events generated in the
 * list component. Implementation required by the ItemListener
* interface.

public void itemStateChanged(ItemEvent e)
{
    String fontStr;
    
    if (e.getItemSelectable().equals(fontList))
    {
        fontStr = fontList.getSelectedIndex();
        fontField.setText(fontStr);
    }
}

/**
 * Allows the applet to listen to events generated in the
 * scrollbar component. Implementation required by the
 * AdjustmentListener interface.
 */

public void adjustmentValueChanged(AdjustmentEvent e)
{
    Integer sizeValue;
    String sizeString;
    
    if (e.getAdjustable().equals(sizeBar))
    {
        sizeValue = new Integer(sizeBar.getValue() * 12);
        sizeString = sizeValue.toString();
        sizeField.setText(sizeString);
    }
}

/**
 * Allows the applet to listen to events generated in the
 * button components. Implementation required by the
 * ActionListener interface.
 */

public void actionPerformed(ActionEvent e)
{
    if (e.getSource().equals(exitButton))
        System.exit(0);
    if (e.getSource().equals(applyButton))
        apply();
}

/**
 * Collects selections of the components in the window in
 * order to change the font style of the text string.
 */

public void apply()
{
    String fontName;
    int fontStyle = 0;
    int weightStyle = 0;
    int slantStyle = 0;
    int fontSize = 0;

    fontName = fontList.getSelectedItem();
    if (fontName.equalsIgnoreCase("Times Roman"))
        fontName = "TimesRoman";
if(weightgrp.getSelectedCheckbox().equals(plainChkbox))
    weightStyle = Font.PLAIN;
if(weightgrp.getSelectedCheckbox().equals(boldChkbox))
    weightStyle = Font.BOLD;

if(slantgrp.getSelectedCheckbox().equals(noslantChkbox))
    slantStyle = 0;
if(slantgrp.getSelectedCheckbox().equals(italicChkbox))
    slantStyle = Font.ITALIC;
fontStyle = weightStyle + slantStyle;

fontSize = (new Integer(sizeField.getText()).intValue);

TextLabel.setFont(new Font(fontName, fontStyle, fontSize));

TextLabel.repaint();

this.repaint();
}
Appendix E

//*******************************************************************
// File: DependencyTraverser.java
// Purpose: Parse the input dependency file and exercise the
// Composite Pattern structure.
//*******************************************************************

package strategy;

// Import the necessary Java packages
import java.io.*;
import java.util.*;
import java.util.StringTokenizer;

/**
 * A class that parses the input dependency file, constructs,
 * and directs the construction of a dependency graph.
 *
 * @author Stephanie Burton
 * @version %I%, %G
 */
public class DependencyTraverser{

    // Attributes
    GraphStrategy _strategy;

    // Methods
    /**
     * Creates a DependencyTraverser object.
     *
     */
    public DependencyTraverser(GraphStrategy theStrategy){
        _strategy = theStrategy;
    }

    /**
     * Parses the input dependency file.
     *
     * @param inFile the name of the input file.
     * @exception IOException if the file does not exist.
     */
    public void ParseMakefile() throws IOException {
        String inFile = "Makefile.txt";
        String line = new String();

        try{
            LineNumberReader inStream = new LineNumberReader
                ( new FileReader (inFile));

            // Parse the input file separating it into
            // single lines
            // to be parsed by a line parser
        }
    }
}
while (line = inStream.readLine() != null) {
    LineParser(line.trim());
}

} catch (IOException e) {
    System.out.println("Makefile.txt does not exist");
    System.exit(1);
}

/**
 * Constructs the composite structure the lines in the
 * input file, one line at a time.
 *
 * @param oneLine a single line of the input file.
 */
private void LineParser(String oneLine) {
    String leaf;
    String internal;
    StringTokenizer line = new StringTokenizer(oneLine, ":,;");
    int tokenCount = line.countTokens();
    if (tokenCount == 1) {  // for Leaf Nodes
        leaf = line.nextToken().trim();
        // Add Node to Graph and Graph List
        System.out.println("Leaf Node encountered: " + leaf);
        _strategy.addNode(leaf);
    } else {  // for Composite Nodes
        internal = line.nextToken().trim();
        // Add Node to graph
        System.out.println("Composite Node encountered: " + internal);
        _strategy.addNode(internal);
        while (line.hasMoreTokens()) {
            int childIndex = -1;
            leaf = line.nextToken().trim();
            // Connect Node to parent node
            System.out.println("Connect " + leaf + " to " + internal);
            _strategy.connectNodes(internal, leaf);
        }
    }
}

/**
 * The main procedure that directs the execution. The expected
 * input file is "Makefile.txt" The format requires all Simple
 * Target to be declared first, followed by declaration of
* Composite
* Targets. The ';' character serves as a line terminator.
* Example of Makefile.txt format:
* <pre>
* Target1:
* Target2:
* Target3:
* Target5:
* Target6:
* Target4: Target1, Target2, Target3;
* Target7: Target5, Target6, Target4, Target1;
* Target8: Target4;
* Target9: Target8, Target7;
* </pre>

* @param args the command line arguments (none required).
* @exception IOException if Makefile.txt does not exist.
*/

public static void main(String[] args) throws IOException {
    GraphStrategy myGraph = null;

    // Parse command line
    if(args.length < 1){
        System.out.println("Representation Option not ");
        System.out.println("specified: ");
        System.out.println("Link or BitSet");
        System.exit(0);
    }

    // Create the specified graph strategy
    if(args[0].equalsIgnoreCase("Link"))
        myGraph = new ListGraphStrategy();
    else if (args[0].equalsIgnoreCase("BitSet"))
        myGraph = new BitSetGraphStrategy();
    else{
        System.out.println("Unrecognized Representation");
        System.exit(0);
    }

    // Create a strategy
    //Graph myGraph = new BitSetGraph();

    // Create an instance of DependencyTraverser
    DependencyTraverser deptrav =
        new DependencyTraverser(myGraph);

    // Parse the input file to build the Composite Structure
    deptrav.ParseMakefile();

    // View adjacent dependents
    System.out.println("Adjacent Neighbors of Target4 are:");
    for (Enumeration e = myGraph.viewNeighbors("Target4");
         e.hasMoreElements();)
    {
        System.out.println(((GraphNode)e.nextElement().getName());
    }
}
// View the shortest path
System.out.println("Adjacent Neighbors of Target4 are:");
for (Enumeration e = myGraph.viewDescendents("Target4");
    e.hasMoreElements()){
    System.out.println(((GraphNode)e.nextElement()).getName());
}

package strategy;

/**
 * A class representing an arbitrary node in a graph.
 * @author Stephanie Burton
 * @version %I%%G%
 */
public class GraphNode{
    // Attributes

    /**
     * The name of the Target.
     */
    String _name;

    /**
     * Flag denoting that node has been visited
     */
    boolean _beenVisited;

    // Methods

    /**
     * Creates a GraphNode with the specified name.
     * @param nameStr the name of the Target.
     */
    public GraphNode(String nameStr){
        _name = nameStr;
        _beenVisited = false;
    }

    /**
     * Returns the name of the GraphNode.
     * @return the name of the Target.
     */
    public String getName(){
        return _name;
    }
}
package strategy;

import java.util.*;

/**
 * A class representing a concrete interface for graphing strategies used to
 * organize GraphNode objects.
 *
 * @author Stephanie Burton
 */
class GraphStrategy {

    // Methods

    /**
     * Creates a GraphStrategy object.
     */
    public GraphStrategy() {}

    /**
     * Adds the a specified node to the graph.
     *
     * @param nodeName string name of the new node.
     */
    public void addNode(String nodeName) {}

    /**
     * Forms a connection between two nodes in the graph.
     *
     * @param origin the name of the node to which a node will be attached.
     * @param destination the name of the node to be attached.
     */
    public void connectNodes(String origin, String destination) {}

    /**
     * Returns an Enumeration of GraphNodes that are directly
     * attached to a specified node in the graph
     *
     * @param parentName the name of the specified node.
     * @return Enumeration of GraphNode objects.
     */
    public Enumeration viewNeighbors(String parentNodeName) {
        // This is a blank vector used to return a dummy enumeration
        Vector temp = new Vector();
        return temp.elements();
    }
}
/**
 * Returns an Enumeration of all GraphNodes that can be reached from
 * a specified node in the graph
 *
 * @param parentName the name of the specified node.
 * @return Enumeration of GraphNode objects.
 */

public Enumeration viewDescendents(String parentName) {
    // This is a blank vector used to return a dummy enumeration
    Vector temp = new Vector();
    return temp.elements();
}

package strategy;

import java.util.*;

/**
 * A class representing a concrete GraphStrategy for representing
 * a directed Graph in the form of a two-dimensional bit array where a
 * '1' at any intersection indicates a connection between the two nodes.
 *
 * @author Stephanie Burton
 */

public class BitSetGraphStrategy extends GraphStrategy {
    // Attributes

    /**
     * A Vector of all nodes in the Graph
     */
    Vector _nodeNames;

    /**
     * A Vector of BitSet representing a two-dimensional bit array.
     */
    Vector _connectionTable;

    // Methods

    /**
     * Creates a BitSetGraphStrategy object.
     */
    public BitSetGraphStrategy() {
        _nodeNames = new Vector();
        _connectionTable = new Vector();
    }
}
/**
 * Adds the a specified node to the graph.
 * @param nodeName string name of the new node.
 */
public void addNode(String nodeName){
    BitSet dependenceSet;
    if (_nodeNames.isEmpty()) {
        // List is empty just add new GraphNode
        _nodeNames.addElement(new GraphNode(nodeName));
        _connectionTable.addElement(new BitSet());
    } else {
        // Check that this Node is not a repeat and add
        for (Enumeration e = _nodeNames.elements();
            e.hasMoreElements();)
            if (((GraphNode)e.nextElement()).getName().equals(nodeName))
                return;
        _nodeNames.addElement(new GraphNode(nodeName));
        _connectionTable.addElement(new BitSet());
    }
}

/**
 * Forms a connection between two nodes in the graph.
 * @param origin the name of the node to which a node will be attached.
 * @param destination the name of the node to be attached.
 */
public void connectNodes(String origin, String destination) {
    GraphNode testNode;
    BitSet tempbs = new BitSet();
    int srcIndex = -1;
    int desIndex = -1;

    // Find the source and destination indexes
    for (Enumeration e = _nodeNames.elements();
        e.hasMoreElements();)
        testNode = (GraphNode)e.nextElement();
    //System.out.println(testNode.getName() + " ");

    if(testNode.getName().equals(origin)){
        srcIndex = _nodeNames.indexOf(testNode);
        //System.out.println(origin);
    }
    if(testNode.getName().equals(destination)){
        desIndex = _nodeNames.indexOf(testNode);
        //System.out.println(destination);
    }
    // Check that Indexes have valid values
if(srcIndex == -1){
    System.out.println("Source Node not defined");
    return;
}

if(desIndex == -1){
    System.out.println("Destination Node not defined");
    return;
}

if(srcIndex == desIndex){
    System.out.println("Node can not be connected to itself");
    return;
}

// make connection (the srcIndex specifies the correct bitset;
// the desIndex specifies the bit in the set to make true)
tempbs = (BitSet)_connectionTable.elementAt(srcIndex);
if(tempbs == null){
    System.out.println("Origin bitset empty");
    return;
}

tempbs.set(desIndex);
_connectionTable.setElementAt(tempbs,srcIndex);
}

/**
 * Returns an Enumeration of GraphNodes that are directly
 * attached to a specified node in the graph
 * @param parentName the name of the specified node.
 * @return Enumeration of GraphNode objects.
 */
public Enumeration viewNeighbors(String parentNodeName) {
    Vector children = new Vector();
    GraphNode testNode;
    int srcIndex = -1;
    BitSet bset;

    for (Enumeration e = _nodeNames.elements();
        e.hasMoreElements(); ) {
        testNode = (GraphNode)e.nextElement();

        if(testNode.getName().equals(parentNodeName))
            srcIndex = _nodeNames.indexOf(testNode);
    }

    if(srcIndex < 0){
        System.out.println("Node not found");
        return children.elements();
    }

    bset = (BitSet)_connectionTable.elementAt(srcIndex);
    if(bset == null){
        System.out.println("Cannot view empty bitset");
    }
public Enumeration viewDescendents(String parentNodeName) {
  Vector temp = new Vector();
  //return temp.elements();
  Vector children = new Vector();
  GraphNode testNode;
  int srcIndex = -1;
  BitSet bset;
  for (Enumeration e = _nodeNames.elements();
       e.hasMoreElements();)
    { testNode = (GraphNode)e.nextElement();
      if (testNode.getName().equals(parentNodeName))
        srcIndex = _nodeNames.indexOf(testNode);
    }
  if (srcIndex < 0) {
    System.out.println("Node not found");
    return children.elements(); //empty vector
  }
  bset = (BitSet)_connectionTable.elementAt(srcIndex);
  if (bset == null) {
    System.out.println("Cannot view empty bitset");
    return children.elements(); //empty vector
  }
  for (int i=0; i<_nodeNames.size(); i++) { if (bset.get(i) == true) {
          testNode = (GraphNode)_nodeNames.elementAt(i);
          children.addElement(testNode);
          Enumeration enum = viewDescendents(testNode.getName());
          if (enum != null) {
            while (enum.hasMoreElements()) {
              GraphNode dep = (GraphNode)enum.nextElement();
            }
          }
        }
  } return children.elements();
}
int exists = 0;
for(Enumeration e = children.elements();
e.hasMoreElements(); ){
    if(((GraphNode)e.nextElement()).getName().equals(dep.getName())){
        exists = 1;
        break;
    }
}

// add if unique
if(exists == 0)
    children.addElement(dep);

return children.elements();

****************************************************************************
** File: ListGraphStrategy.java
** Purpose: Concrete Graph class for organizingGraphNode objects
** in a Graph into a List of Vector.
******************************************************************************

package strategy;
import java.util.*;

/**
 * A class representing a concrete Graph for representing a directed
 * Graph in the form of a List of Vectors.
 *
 * @author Stephanie Burton
 */
public class ListGraphStrategy extends GraphStrategy{
    // Attributes

    /**
     * A Hashtable representing the Graph. (String, Vector of GraphNodes)
     */
    Hashtable _nodeTable;

    // Methods

    /**
     * Creates a ListGraphStrategy object.
     */
    public ListGraphStrategy() {
        _nodeTable = new Hashtable();
    }
/**
 * Adds the specified node to the graph.
 *
 * @param  nodeName string name of the new node.
 */
public void addNode(String nodeName) {
    Vector nodeList;
    GraphNode node;
    String keyName = null;

    // System.out.println("Add Node");
    if(!_nodeTable.isEmpty())
        System.out.println("List Empty");
    else {
        for (Enumeration e = _nodeTable.keys();
            e.hasMoreElements();)
            keyName = (String)e.nextElement();
        if (keyName.equals(nodeName))
            break;
    }
    keyName = null;

    if (keyName == null) {
        keyName = new String(nodeName);

        // Create new Graph Node with specified name
        node = new GraphNode(nodeName.trim());

        // Create a vector to hold graph nodes
        nodeList = new Vector();

        // Add GraphNode to list
        nodeList.addElement(node);

        // Insert name and vector pair into hashtable
        _nodeTable.put(keyName, nodeList);

        // Debug: verify nodes have been added to the list
        // System.out.println("DEBUG: ");
        Vector v = (Vector)_nodeTable.get(keyName);
        if (v != null) {
            GraphNode g = (GraphNode)v.firstElement();
            System.out.println("Added Node: " + g.getName());
        }
    }
    else
        System.out.println("Redeclaration of node: " + keyName);
}

/**
 * Forms a connection between two nodes in the graph.
 *
 * @param origin the name of the node to which a node will be attached.
 */
* @param destination the name of the node to be attached.
* /
public void connectNodes(String origin, String destination) {
    String key = null;
    GraphNode childNode;
    Vector valueVector;

    // Create the childNode
    childNode = new GraphNode(destination);

    // Find the correct child and add the child to its
    // vector of dependencies
    for (Enumeration e = _nodeTable.keys(); e.hasMoreElements();)
    {
        key = (String)e.nextElement();
        if(key.equals(origin)){
            //System.out.println(key);
            break;
        }
    }  
    key = null;
}

try{
    //key = containsKey(origin);
    valueVector = (Vector)_nodeTable.get(key);
    valueVector.addElement(childNode);
    _nodeTable.put(key,valueVector);
}
catch(NullPointerException e){
    System.out.println("ConnectNode: "+origin+" not in table");
    System.exit(0);
}

/**
 * Returns an Enumeration of GraphNodes that are directly
 * attached to a specified node in the graph.
 *
 * @param parentName the name of the specified node.
 * @return Enumeration of GraphNode objects.
 */
public Enumeration viewNeighbors(String parentName){
    String key = null;
    Vector valueVector = new Vector();

    for (Enumeration e = _nodeTable.keys(); e.hasMoreElements();)
    {
        key = (String)e.nextElement();
        if(key.equals(parentName)){
            //System.out.println(key);
            break;
        }
    }  
    key = null;

    if(key != null){

}
for (Enumeration e = ((Vector)_nodeTable.get(key)).elements();
   e.hasMoreElements() ; ) {
   valueVector.addElement(e.nextElement());
}
valueVector.removeElementAt(0);
}
else{
   System.out.println(parenfName + " not in table");
   System.exit(0);
}
return (valueVector.elements());

/**
 * Returns an Enumeration of all GraphNodes that can be reached
 * from a specified node in the graph
 *
 * @param parentNode the name of the specified node.
 * @return Enumeration of GraphNode objects.
 *
 */
public Enumeration viewDescendents(String parentName) {
String key = null;
Vector valueVector = new Vector();
Vector temp = new Vector();

for (Enumeration e = _nodeTable.keys(); e.hasMoreElements();)
   key = (String)e.nextElement();
if(key.equals(parentName)) {
   //System.out.println("KEY NAME: " + key);
   break;
}
key = null;

if(key != null) {
   valueVector = (Vector)_nodeTable.get(key);
}
else {
   System.out.println(parentName + " not in table");
   System.exit(0);
}

if(valueVector.size() == 1) { // leaf node
   return temp.elements(); // return a null vector
}
else {
   // Show List for Debug
   //for (Enumeration e = list.elements(); e.hasMoreElements(); ) {
   //   System.out.println(((GraphNode)e.nextElement()).getName());
   //}
   //System.out.println("--");
   
   //System.exit(0);
}
for(int i=1; i<valueVector.size(); i++){
    GraphNode depNode = (GraphNode)valueVector.elementAt(i);
    temp.addElement(depNode);
    Enumeration enum = viewDescendents(depNode.getName());
    if(enum != null){
        while(enum.hasMoreElements()){
            GraphNode dep = (GraphNode)enum.nextElement();
            // check for redundancy
            int exists = 0;
            for (Enumeration e = temp.elements(); e.hasMoreElements(); ) {
                if(((GraphNode)e.nextElement().getName().equals(dep.getName()))){
                    exists = 1;
                    break;
                }
            }
            // add if unique
            if(exists == 0)
                temp.addElement(dep);
        }
    }
}

return temp.elements();  //Vector should contain something
APPENDIX F Template Method Application Source Code
Appendix F

package templateMethod;

//Import the necessary Java packages
import java.io.*;
import java.util.*;
import java.util.StringTokenizer;

/**
 * A class that parses the input dependency file, constructs,
 * and directs construction of a dependency graph.
 *
 * @author Stephanie Burton
 * @version %I%, %G
 */
public class DependencyTraverser{
    // Attributes
    Graph _strategy;
    // Methods
    /**
     * Creates a DependencyTraverser object.
     */
    public DependencyTraverser(Graph theStrategy){
        _strategy = theStrategy;
    }
    /**
     * Parses the input dependency file.
     *
     * @param inFile the name of the input file.
     * @exception IOException
     * if the file does not exist.
     */
    public void ParseMakefile() throws IOException {
        String inFile = "Makefile.txt";
        String line = new String();

        // Open the input file, catch the exception
        // here(error checking) if the file does not exist,
        // stop here.
        try{
            LineNumberReader inStream = new LineNumberReader
                (new FileReader(inFile));

            // Parse the input file separating it into single lines
            // to be parsed by a line parser
            while ((line = inStream.readLine()) != null){

LineParser(line.trim());
}

catch(IOException e){
  System.out.println("Makefile.txt does not exist");
  System.exit(1);
}

/**
 * Constructs the composite structure the lines in the
 * input file, one line at a time.
 *
 * @param oneLine a single line of the input file.
 */
private void LineParser(String oneLine){
  String leaf;
  String internal;
  StringTokenizer line = new StringTokenizer(oneLine, ":,;");
  int tokenCount = line.countTokens();
  if (tokenCount == 1) { // for Leaf Nodes
    leaf = line.nextToken().trim();
    // Add Node to Graph and Graph List
    System.out.println("Leaf Node encountered: "+ leaf);
    _strategy.insertNode(leaf);
  }
  else { // for Composite Nodes
    internal = line.nextToken().trim();
    // Add Node to Graph
    System.out.println("Composite Node encountered: "+
    internal);
    _strategy.insertNode(internal);
    while (line.hasMoreTokens()) {
      int childIndex = -1;
      leaf = line.nextToken().trim();
      // Connect Node to parent node
      System.out.println("Connect " + leaf + " to "+
      internal);
      _strategy.connectNodes(internal, leaf);
    }
  }
}

/**
 * The main procedure that directs the execution. The expected
 * input file is "Makefile.txt". The format requires all Simple
 * Target to be declared first, followed by declaration of Composite
 * Targets. The ',' character serves as a line terminator.
 */
Example of Makefile.txt format:

```makefile
Target1;
Target2;
Target3;
Target4;
Target5;
Target6;
Target7: Target1, Target2, Target3;
Target8: Target5, Target6, Target4, Target1;
Target9: Target8, Target7;
```

* @param args the command line arguments (none required).
* @exception IOException
*   if Makefile.txt does not exist.
* /
public static void main(String[] args) throws IOException {
    Graph myGraph = null;

    // Parse command line
    if(args.length < 1){
        System.out.print("Representation Option ");
        System.out.print("not specified: ");
        System.out.println("Link or BitSet");
        System.exit(0);
    }

    // Create the specified graph strategy
    if(args[0].equalsIgnoreCase("List"))
        myGraph = new ListGraph();
    else if (args[0].equalsIgnoreCase("BitSet"))
        myGraph = new BitSetGraph();
    else{
        System.out.println("Unrecognized Representation");
        System.exit(0);
    }

    // Create an instance of DependencyTraverser
    DependencyTraverser deptrav =
        new DependencyTraverser(myGraph);

    // Parse the input file to build the Composite Structure
    deptrav.ParseMakefile();

    // View adjacent dependents
    System.out.println("Adjacent Neighbors of Target1 are:");
    for (Enumeration e = myGraph.viewNeighbors("Target9");
         e.hasMoreElements();
         System.out.println(((GraphNode)e.nextElement()).getName());
    }

    // View shortest path
    System.out.println("Shortest path from Target8 to Target9: ");
    Vector path = myGraph.shortestPath("Target9", "Target1");
    if(path.size() == 0){
        System.out.print("Destination could not be reached");
    }
System.out.println(" from the source");

} else {
    System.out.println("Path Size = " + path.size());
    System.out.println("From Target");
    for (int i = 0; i < path.size(); i++)
        System.out.println("to " +
                         ((GraphNode) path.elementAt(i)).getName());
}

} //**********************************************************************

// File: GraphNode.java
// Purpose: Defines the template for all GraphNodes.
//**********************************************************************

package templateMethod;

/**
 * A class representing an arbitrary node in a graph.
 *
 * @author Stephanie Burton
 * @version %I%, %G%
 */

public class GraphNode {
    // Attributes

    /**
     * The name of the Target.
     *
     */
    String _name;

    /**
     * Flag denoting that node has been visited
     *
     */
    boolean _beenVisited;

    // Methods

    /**
     * Creates a GraphNode with the specified name.
     *
     * @param nameStr the name of the Target.
     */
    public GraphNode(String nameStr) {
        _name = nameStr;
        _beenVisited = false;
    }

    /**
     * Returns the name of the GraphNode.
     *
     */
    public String getName() {
        return _name;
    }

    //...
public String getName(){
    return _name;
}

package templateMethod;

import java.util.*;

/**
 * An abstract class representing the interface of
 * a singly directed graph data structure.
 * 
 * @author  Stephanie Burton
 */
abstract class Graph{

    // Methods
    
    /**
     * Constructor for the abstract Graph class. An abstract
     * can not be directly instantiated but must be subclassed.
     */
    public Graph() {
    }

    /**
     * An abstract declaration of method to add nodes to a graph.
     * 
     * @param  nodeName  the name of the node to be added to the graph.
     */
    abstract void insertNode(String nodeName);

    /**
     * An abstract declaration of method to connect two nodes in a graph.
     * The connection is directed from the origin node to the destination node.
     * 
     * @param  origin  the node name of the source to which a node is connected.
     * @param  destination  the node name of the destination node to be connected.
     */
    abstract void connectNodes(String origin, String destination);

    /**
     * An abstract declaration of a method that returns list of all the names
     * of nodes adjacent to the specified node in the graph.
     * 
     * @param  parentNode  the name of the specified node.
     * @return  an Enumeration of the GraphNodes.
     */
    abstract Enumeration viewNeighbors(String parentNodeName);

/**
 * Returns a vector of GraphNodes that represent the shortest path
 * from a source node to a destination node. The source node is
 * not included in the Vector.
 *
 * @param origin the starting node.
 * @param destination the ending node.
 * @return A vector of GraphNode objects.
 */

public final Vector shortestPath(String origin, String destination) {
    Vector path = null;
    Vector bestPath = null;
    GraphNode neighbor = null;
    int shortestPath = 0;

    // Check if node has any neighbors
    Enumeration enum = viewNeighbors(origin);
    /*if (enum == null) {
        System.out.println("End of Graph Reached: ");
        System.out.println(origin + " has not neighbor.");
        return path; // path is null;
    }*/

    // new Vector to hold the result
    path = new Vector();
    bestPath = new Vector();

    // Check if destination is an adjacent neighbor
    while (enum.hasMoreElements()) {
        neighbor = (GraphNode) enum.nextElement();
        if (!neighbor.getName().equals(destination)) {
            path.addElement(neighbor);
            return path;
        }
    }

    for (Enumeration enumr = viewNeighbors(origin);
         enumr.hasMoreElements(); ) {
        GraphNode descendent = (GraphNode) enumr.nextElement();
        path = shortestPath(
            descendent.getName(), destination);
        if ((path == null) || (path.size() == 0))
            continue;
        if ((bestPath.size() == 0) ||
            (bestPath.size() > path.size())) {
            bestPath = path;
            bestPath.insertElementAt(descendent, 0);
        }
    }

    return bestPath;
}
package templateMethod;

import java.util.*;

/**
 * A class representing a concrete Graph class for representing a directed Graph in the form of a two-dimensional bit array where a '1' at any intersection indicates a connection between the two nodes.
 * @author Stephanie Burton
 */
public class BitSetGraph extends Graph{
  // Attributes
  /**
   * A Vector of all nodes in the Graph
   */
  Vector _nodeNames;

  /**
   * A Vector of BitSet representing a two-dimensional bit array.
   */
  Vector _connectionTable;

  // Methods

  /**
   * Creates a BitSetGraph object.
   */
  public BitSetGraph() {
    _nodeNames = new Vector();
    _connectionTable = new Vector();
  }

  /**
   * Adds the a specified node to the graph.
   *
   * @param nodeName string name of the new node.
   */
  public void insertNode(String nodeName){
    BitSet dependenceSet;
    if (_nodeNames.isEmpty()){
      // List is empty just add newGraphNode
      _nodeNames.addElement(new GraphNode(nodeName));
      _connectionTable.addElement(new BitSet());
    }
  }
}
{  
  // Check that this Node is not a repeat and add  
  for (Enumeration e = _nodeName.elements();  
      e.hasMoreElements(); ) {  
    if(((GraphNode)e.nextElement()).getName().equals(nodeName))  
      return ;  
  }  
  _nodeName.addElement(new GraphNode(nodeName));  
  _connectionTable.addElement(new BitSet());  
}

/**  
* Forms a connection between two nodes in the graph.  
*  
* @param origin the name of the node to which a node will be attached.  
* @param destination the name of the node to be attached.  
*/  
public void connectNodes(String origin, String destination) {  
  GraphNode testNode;  
  BitSet tempbs = new BitSet();  
  int srcIndex = -1;  
  int desIndex = -1;  

  // Find the source and destination indexes  
  for (Enumeration e = _nodeName.elements();  
      e.hasMoreElements(); ) {  
    testNode = (GraphNode)e.nextElement();  
    //System.out.println(testNode.getName() + "]");  

    if(testNode.getName().equals(origin)))  
      srcIndex = _nodeName.indexOf(testNode);  
    //System.out.println(origin);  
  }  

  if(testNode.getName().equals(destination))  
    desIndex = _nodeName.indexOf(testNode);  
    //System.out.println(destination);  

  // Check that Indexes have valid values  
  if(srcIndex == -1){  
    System.out.println("Source Node not defined");  
    return ;  
  }

  if(desIndex == -1){  
    System.out.println("Destination Node not defined");  
    return ;  
  }

  if(srcIndex == desIndex){  
    System.out.println("Node can not be ");  
    System.out.println("connected to itself");  
    return ;  
  }

  }
// make connection (the srcIndex specifies the correct
// bitset; the desIndex specifies the bit in the set to
// make true)
tempbs = (BitSet)_connectionTable.elementAt(srcIndex);
if(tempbs == null){
    System.out.println("Origin bitset empty");
    return;
}

    tempbs.set(desIndex);
    _connectionTable.setElementAt(tempbs,srcIndex);

/**
   * Returns an Enumeration of GraphNodes that are attached
   * a specified node in the graph
   *
   * @param parentNodeName the name of the specified node.
   *
   * @return Enumeration of GraphNode objects.
   *
   */
public Enumeration viewNeighbors(String parentNodeName)
{
    Vector children = new Vector();
    GraphNode testNode;
    int srcIndex = -1;
    BitSet bset;

    for (Enumeration e = _nodeNames.elements();
    e.hasMoreElements() ) {
        testNode = (GraphNode)e.nextElement();

        if(testNode.getName().equals(parentNodeName))
            srcIndex = _nodeNames.indexOf(testNode);
    }

    if(srcIndex < 0){
        System.out.println("Node not found");
        System.exit(0);
    }

    bset =
        (BitSet)_connectionTable.elementAt(srcIndex);

    if(bset == null){
        System.out.println("Cannot view empty bitset");
        return children.elements();
    }

    for(int i=0; i<_nodeNames.size(); i++){
        if(bset.get(i) == true)
            children.addElement(_nodeNames.elementAt(i));
    }

    return children.elements();
package templateMethod;

import java.util.*;

/**
 * A class representing a concrete Graph class for representing a directed
 * Graph in the form of a List of Vectors.
 *
 * @author Stephanie Burton
 */
public class ListGraph extends Graph{
    // Attributes

    /**
     * A Hashtable representing the Graph. (String, Vector of GraphNodes)
     */
    Hashtable _nodeTable;

    // Methods

    /**
     * Creates a ListGraph object.
     */
    public ListGraph() {
        _nodeTable = new Hashtable();
    }

    /**
     * Adds the a specified node to the graph.
     *
     * @param nodeName string name of the new node.
     */
    public void insertNode(String nodeName) {
        Vector nodeList;
        GraphNode node;
        String keyName = null;

        //System.out.println("Add Node");
        if(_nodeTable.isEmpty())
            System.out.println("List Empty");
        else {
            Enumeration e = _nodeTable.keys();
            while (e.hasMoreElements()) {
                keyName = (String)e.nextElement();
                if(keyName.equals(nodeName))
                    return;
            }
            nodeList = (Vector)_nodeTable.get(nodeName);
            if(nodeList == null) { /* create a new entry */
                nodeList = new Vector();
                _nodeTable.put(nodeName, nodeList);
            }
            node = new GraphNode(nodeName);
            nodeList.addElement(node);
        }
    }

    public void deleteNode(String nodeName) {
        Vector nodeList = (Vector)_nodeTable.get(nodeName);
        if(nodeList != null) {
            nodeList.removeElement(nodeName);
            if(nodeList.isEmpty())
                _nodeTable.remove(nodeName);
            else
                System.out.println("Delete Node "+nodeName);
        }
    }

    public GraphNode getNode(String nodeName) {
        Vector nodeList = (Vector)_nodeTable.get(nodeName);
        if(nodeList != null) {
            GraphNode node = (GraphNode)nodeList.elementAt(0);
            return node;
        }
        return null;
    }

    public void print() {
        Enumeration e = _nodeTable.keys();
        while (e.hasMoreElements()) {
            String keyName = (String)e.nextElement();
            Vector nodeList = (Vector)_nodeTable.get(keyName);
            System.out.println("Number of Nodes in "+keyName+":");
            for (Enumeration e1 = nodeList.elements(); e1.hasMoreElements();)
                System.out.println("Node "+e1.nextElement());
        }
    }
}

F-10
break;
}  
keyName = null;
}

if(keyName == null){
keyName = new String(nodeName);

// Create new Graph Node with specified name
node = new GraphNode(nodeName.trim());

// Create a vector to hold graph nodes
nodeList = new Vector();

// Add GraphNode to list
nodeList.addElement(node);

// Insert name and vector pair into hashtable
_nodeTable.put(keyName, nodeList);

// Debug: verify nodes have been added to the list
/*@System.out.println("DEBUG: ");
Vector v = (Vector) _nodeTable.get(keyName);
if (v != null){
    GraphNode g = (GraphNode)v.firstElement();
    System.out.println("Added Node: ");
    System.out.println(g.getName());
}*/
}
else

System.out.println("Redeclaration of node: ");
System.out.println(keyName);
}

/**
 * Forms a connection between two nodes in the graph.
 *
 * @param  origin the name of the node to which a node will be attached.
 * @param  destination the name of the node to be attached.
 */
public void connectNodes(String origin, String destination) {
    String key = null;
    GraphNode childNode;
    Vector valueVector;

    // Create the childNode
    childNode = new GraphNode(destination);

    // Find the correct the parent and add the child to its
    // vector of dependencies
    for (Enumeration e = _nodeTable.keys(); e.hasMoreElements();){
        key = (String)e.nextElement();
        if(key.equals(origin)){
            //System.out.println(key);
            break;
        }
key = null;
}
try{
    //key = containsKey(origin);
    valueVector = (Vector)_nodeTable.get(key);
    valueVector.addElement(childNode);
    _nodeTable.put(key,valueVector);
} catch(NullPointerException e){
    System.out.print("ConnectNode: " + origin);
    System.out.println(" not in table");
    System.exit(0);
}

/**
 * Returns an Enumeration of GraphNodes that are attached
 * a specified node in the graph
 *
 * @param  parentName the name of the specified node.
 * @return  Enumeration of GraphNode objects.
 *
 */
public Enumeration viewNeighbors(String parentName) {
    String key = null;
    Vector valueVector = new Vector();
    for (Enumeration e = _nodeTable.keys();
         e.hasMoreElements();)
    {
        key = (String)e.nextElement();
        if(key.equals(parentName)){
            //System.out.println(key);
            break;
        }
    }
    key = null;
}
if(key != null){
    for (Enumeration e = ((Vector)_nodeTable.get(key)).elements();
         e.hasMoreElements();)
    {
        valueVector.addElement(e.nextElement());
    }
    valueVector.removeElementAt(0);
}
else {
    System.out.println(parentName + " not in table");
    System.exit(0);
}
return (valueVector.elements());
}
Appendix G

import java.awt.*;
import java.util.Vector;

/**
 * A class representing a Facade that provides a simple interface
 * that hides the complex interaction of the underlying subsystem
 * of classes.
 *
 * @author Stephanie Burton
 */
public class GraphFacade {

    /**
     * The Graph object associated with this Facade.
     *
     */
    Graph theGraph;

    /**
     * The Request object associated with this Facade.
     *
     */
    Request userRequest;

    /**
     * The Dialog object associated with this Facade, used to gather user request.
     *
     */
    UserInputGUI inputDialog;

    /**
     * Creates a Graph Facade object.
     *
     * @exception IOException
     * if the expected input file does not exist.
     */
    GraphFacade() throws java.io.IOException {
        init();
    }

    /**
     * Sets the values of the class variables.
     *
     * @exception IOException
     * if the expected input file does not exist.
     */
    private void init() throws java.io.IOException{
/Build Graph
theGraph = new ListGraph();
DependencyTraverser deptrav = new DependencyTraverser(theGraph); deptrav.ParseMakefile();

//Construct Dialog box
Frame dialogFrame = new Frame();
inputDialog = new UserInputGUI(dialogFrame, theGraph.getNodeNameList());
}

/**
 * Returns an appropriate Request object based on the
 * user request information collected in the Dialog.
 *
 * @return one of the available derived Request objects.
 */
public Request getRequest(){

    inputDialog.show();

    if(inputDialog.getRequestStr().equals("AllPaths")){
       System.out.println("AllPaths Request");
       userRequest = new AllPathsRequest( 
            inputDialog.getSource(), inputDialog.getDestination(),
            theGraph);
    }
    if(inputDialog.getRequestStr().equals("ShortestPath")){
       System.out.println("ShortestPath Request");
       userRequest = new ShortestPathRequest( 
            inputDialog.getSource(), inputDialog.getDestination(),
            theGraph);
    }
    if(inputDialog.getRequestStr().equals("Exit")) {
       System.out.println("Exit Request");
       userRequest = new ExitRequest();
    }

    return userRequest;
}

/**
 * Main method to start the GraphFacade application
 *
 * @param args command line arguments; none required
 * @exception IOException
 * if the expected input file does not exist.
 */
public static void main(String[] args) throws java.io.IOException{

    // Create a Facade
    GraphFacade f = new GraphFacade();

    while(true){

        // Get User Request
        Request r = f.getRequest();
// Process Request
r.doRequest();

}

}

}


// File: DependencyTraverser.java
// Purpose: Parse the input dependency file and exercise the
// Composite Pattern structure.
// Import the necessary Java packages
import java.io.*;
import java.util.*;
import java.util.StringTokenizer,
/**
 * A class that parses the input dependency file, constructs,
 * and directs the construction of a dependency graph.
 * @author Stephanie Burton
 * @version %I%, %G
 */
public class DependencyTraverser{
   // Attributes
   Graph _strategy;

   // Methods
   /**
    * Creates a DependencyTraverser object.
    */
   public DependencyTraverser(Graph theStrategy){
      _strategy = theStrategy;
   }

   /**
    * Parses the input dependency file.
    *
    * @param inFile the name of the input file.
    * @exception IOException
    * if the file does not exist.
    */
   public void ParseMakefile() throws IOException {
      String inFile = "Makefile.txt";

      String line = new String();

      // Open the input file, catch the exception
      // here(error checking) if the file does not exist,
      // stop here.
      try{ 
         LineNumberReader inStream = new LineNumberReader
            (new FileReader (inFile));
   }

}
// Parse the input file separating it into single lines
// to be parsed by a line parser
while ( (line = inStream.readLine()) != null) {
  LineParser(line.trim());
}

} catch (IOException e) {
  System.out.println("Makefile.txt does not exist");
  System.exit(1);
}

/**
 * Constructs the composite structure the lines in the
 * input file, one line at a time.
 *
 * @param oneLine a single line of the input file.
 *
 */
private void LineParser(String oneLine) {
  String leaf;
  String internal;
  StringTokenizer line = new StringTokenizer(oneLine, ":,;");
  int tokenCount = line.countTokens();

  if (tokenCount == 1) { // for Leaf Nodes
    leaf = line.nextToken().trim();
    // Add Node to Graph and Graph List
    System.out.println("Leaf Node encountered: " + leaf);
    _strategy.insertNode(leaf);
  } else { // for Composite Nodes
    internal = line.nextToken().trim();
    // Add Node to graph
    System.out.println("Composite Node encountered: " + internal);
    _strategy.insertNode(internal);

    while (line.hasMoreTokens()) {
      int childIndex = -1;
      leaf = line.nextToken().trim();
      // Connect Node to parent node
      System.out.println("Connect " + leaf + " to " + internal);
      _strategy.connectNodes(internal, leaf);
    }
  }
}
* The main procedure that directs the execution. The expected
  input file is "Makefile.txt" The format requires all Simple
  Target to be declared first, followed by declaration of Composite
  Targets. The ';' character serves as a line terminator.

* Example of Makefile.txt format:

```<pre>*
* Target1;
* Target2;
* Target3;
* Target5;
* Target6;
* Target4: Target1, Target2, Target3;
* Target7: Target5, Target6, Target4, Target1;
* Target8: Target4;
* Target9: Target8, Target7;
* </pre>*

* @param args the command line arguments (none required).
* @exception IOException
*     if Makefile.txt does not exist.
*/

public static void main(String[] args) throws IOException {
    Graph myGraph = null;

    myGraph = new ListGraph();

    // Create an instance of DependencyTraverser
    DependencyTraverser deptrav =
        new DependencyTraverser(myGraph);

    // Parse the input file to build the Composite Structure
    deptrav.ParseMakefile();

    // View adjacent dependents
    System.out.println("Adjacent Neighbors of Target1 are:");
    for (Enumeration e = myGraph.viewNeighbors("Target9");
        e.hasMoreElements();)
    {
        System.out.println(((GraphNode)e.nextElement()).getName());
    }

    // View shortest path
    System.out.println("Shortest path from Target8 to Target6:");
    Vector path = myGraph.shortestPath("Target9", "Target1");
    if(path.size() == 0){
        System.out.println("Destination could not be reached");
        System.out.println(" from the source");
    }
    else{
        System.out.println("Path Size = " + path.size());
        System.out.println("From Target9");
        for(int i = 0; i<path.size(); i++)
        System.out.println("to " + ((GraphNode)path.elementAt(i)).getName());
    }
}

```
public class GraphNode {
    // Attributes
    String _name;
    boolean _beenVisited;

    // Methods
    public GraphNode(String nameStr) {
        _name = nameStr;
        _beenVisited = false;
    }

    public String getName() {
        return _name;
    }
}
import java.util.*;

/**
 * An abstract class representing the interface of
 * a singly directed graph data structure.
 */
abstract class Graph {

    // Methods

    /**
     * Constructor for the abstract Graph class. An abstract
     * can not be directly instantiated but must be subclassed.
     */
    public Graph() {
    }

    /**
     * An abstract declaration of method to add nodes to a graph.
     */
    abstract void insertNode(String nodeName);

    /**
     * An abstract declaration of method to connect two nodes in a graph.
     * The connection is directed from the origin node to the destination node.
     */
    abstract void connectNodes(String origin, String destination);

    /**
     * An abstract declaration of a method that returns list of all the names
     * of nodes in the graph.
     */
    abstract Enumeration getNodeNameList();

    /**
     * An abstract declaration of a method that returns list of all the names
     * of nodes adjacent to the specified node in the graph.
     */
    abstract Enumeration viewNeighbors(String parentNodeName);
/** *
* Returns a vector of GraphNodes that represent the shortest path 
* from a source node to a destination node. The source node is 
* not included in the Vector. 
* *
* @param origin the starting node. 
* @param destination the ending node. 
* @return A vector of GraphNode objects. 
*/
public Vector shortestPath(String origin, String destination){
    Vector path = null;
    Vector bestPath = null;
    GraphNode neighbor = null;
    int shortestpath =

    // Check is node has any neighbors
    Enumeration enum = viewNeighbors(origin);
    // new Vector to hold the result
    path = new Vector();
    bestPath = new Vector();

    // Check if destination is an adjacent neighbor
    while(enum.hasMoreElements()){
        neighbor = (GraphNode)enum.nextElement();
        if((neighbor.getName().equals(destination))){
            path.addElement(neighbor);
            return path;
        }
    }

    for(Enumeration enumr = viewNeighbors(origin);
        enumr.hasMoreElements();){
        GraphNode descendent = (GraphNode)enumr.nextElement();
        path = shortestPath( 
            descendent.getName(), destination);
        if((path == null) || (path.size() == 0))
            continue;
        if( (bestPath.size() == 0) || 
            (bestPath.size() > path.size()) ){
            bestPath = path;
            bestPath.insertElementAt(descendent,0);
        }
    }

    return bestPath;
}

/** *
* List all paths from a source node to a destination node. 
* An empty vector is included to serve as memory for the path 
* traveled in order to print out the nodes during traversal, however, 
* by the end of the traversal the vector will be empty again. 
*/
public void allPaths(String origin, String destination, Vector scratchPad) {

    GraphNode neighbor = null;

    // add yourself to the path to remember
    scratchPad.addElement(origin);

    // Check is node has any neighbors
    Enumeration enum = viewNeighbors(origin);

    // Check if destination is an adjacent neighbor
    while (enum.hasMoreElements()) {
        neighbor = (GraphNode) enum.nextElement();
        if (neighbor.getName().equals(destination)) {
            // your name to the list
            scratchPad.addElement(neighbor.getName());

            // print out the path
            for (int i = 0; i < scratchPad.size(); i++) {
                System.out.print((String) scratchPad.elementAt(i) + " ");
            }
            System.out.println(" ");

            // Remove last element before returning
            scratchPad.removeElement((String) scratchPad.lastElement());
        }
    }

    for (Enumeration enumr = viewNeighbors(origin);
         enumr.hasMoreElements(); ) {
        GraphNode descendent = (GraphNode) enumr.nextElement();
        allPaths(descendent.getName(), destination, scratchPad);

        // Remove last element before returning
        scratchPad.removeElement((String) scratchPad.lastElement());
    }
}

//****************************************************************************
// File: ListGraph.java
// Purpose: Concrete Graph class for organizing GraphNode objects in a Graph into a List of Vector.
//****************************************************************************
import java.util.*;

/**
 * A class representing a concrete Graph class for representing a directed
 * Graph in the form of a List of Vectors.
 */

@author Stephanie Burton

public class ListGraph extends Graph {
    // Attributes

    /**
     * A Hashtable representing the Graph.(String, Vector of GraphNodes)
     */
    Hashtable _nodeTable;

    // Methods

    /**
     * Creates a ListGraph object.
     */
    public ListGraph() {
        _nodeTable = new Hashtable();
    }

    /**
     * Adds the a specified node to the graph by adding a new pair
     * to the nodeTable (String name of the new node, A Vector to
     * store GraphNodes connected to this node).
     *
     * @param nodeName string name of the new node.
     */
    public void insertNode(String nodeName) {
        Vector nodeList;
        GraphNode node;
        String keyName = null;

        //System.out.println("Add Node");

        if(_nodeTable.isEmpty())
            System.out.println("List Empty");
        else {
            for (Enumeration e = _nodeTable.keys() ;
                 e.hasMoreElements();)
            {
                keyName = (String)e.nextElement();
                if(keyName.equals(nodeName))
                    break;
            }
            keyName = null;
        }

        if(keyName == null){
            keyName = new String(nodeName);

            // Create new Graph Node with specified name
            node = new GraphNode(nodeName.trim());
// Create a vector to hold graph nodes
nodeList = new Vector();

// Add GraphNode to list
nodeList.addElement(node);

// Insert name and vector pair into hashtable
_nodeTable.put(keyName, nodeList);

// Debug: verify nodes have been added to the list
/*System.out.println("DEBUG: ");
Vector v = (Vector) _nodeTable.get(keyName);
if (v != null) {
    GraphNode g = (GraphNode)v.firstElement();
    System.out.println("Added Node: ");
    System.out.println(g.getName());
}*/
}
else
    System.out.println("Redeclaration of node: ");
System.out.println(keyName);

/**
* Returns an Enumeration of strings representing the names
* of all the nodes in the graph.
* @return Enumeration of Strings.
*/
public Enumeration getNodeNameList() {
    return _nodeTable.keys();
}

/**
* Forms a connection between two nodes in the graph.
* @param origin the name of the node to which a node will be attached.
* @param destination the name of the node to be attached.
*/
public void connectNodes(String origin, String destination) {
    String key = null;
    GraphNode childNode;
    Vector valueVector;
    // Create the childNode
    childNode = new GraphNode(destination);
    // Find the correct the parent and add the child to its
    // vector of dependencies
    for (Enumeration e = _nodeTable.keys();
         e.hasMoreElements()) {
        key = (String)e.nextElement();
        if (key.equals(origin)) {
            //System.out.println(key);
            break;
        }
}
key = null;

try {
    //key = containsKey(origin);
    valueVector = (Vector)_nodeTable.get(key);
    valueVector.addElement(childNode);
    _nodeTable.put(key, valueVector);
} catch (NullPointerException e) {
    System.out.print("ConnectNode: " + origin);
    System.out.println(" not in table");
    System.exit(0);
}

/**
 * Returns an Enumeration of GraphNodes that are attached
 * a specified node in the graph
 *
 * @param parentName the name of the specified node.
 * @return Enumeration of GraphNode objects.
 */
public Enumeration viewNeighbors(String parentName) {
    String key = null;
    Vector valueVector = new Vector();

    for (Enumeration e = _nodeTable.keys(); e.hasMoreElements();)
    {
        key = (String)e.nextElement();
        if (key.equals(parentName)) {
            //System.out.println(key);
            break;
        }
        key = null;
    }

    if (key != null) {
        for (Enumeration e = ((Vector)_nodeTable.get(key)).elements();
            e.hasMoreElements();)
        {
            valueVector.addElement(e.nextElement());
        }
        valueVector.removeElementAt(0);
    }
    else {
        System.out.println(parentName + " not in table");
        System.exit(0);
    }

    return (valueVector.elements());
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
import java.util.*;

/**
 * A class representing a Dialog box for collecting user requests.
 *
 * @author Stephanie Burton
 */
public class UserInputGUI extends Dialog implements ActionListener, ItemListener {

    /**
     * TextField to display the name of the source name.
     *
     */
    private TextField source;

    /**
     * TextField to display the name of the destination name.
     *
     */
    private TextField destination;

    /**
     * List containing valid source names.
     *
     */
    private List sourceList;

    /**
     * List containing valid destination names.
     *
     */
    private List destList;

    /**
     * OK button signalling the end of user input.
     *
     */
    private Button okButton;

    /**
     * Checkbox group of valid user requests.
     *
     */
    private CheckboxGroup requestGroup;

    /**
     * Checkbox for requesting a search for AllPaths.
     */
}
private Checkbox allChkbox;

/**
 * Checkbox for requesting a search for the ShortestPath.
 */
private Checkbox shortChkbox;

/**
 * Checkbox for requesting to exit the application.
 */
private Checkbox exitChkbox;

/**
 * String to store the chosen source string.
 */
private String sourceStr = new String();

/**
 * String to store the chosen destination string.
 */
private String destnStr = new String();

/**
 * String to store the chosen request in string form.
 */
private String requestStr = new String();

/**
 * Creates a UserInputGUI object.
 *
 * @param parent parent Frame for the dialog box.
 * @param choices Enumeration of strings for valid choices.
 */
UserInputGUI(Frame parent, Enumeration choices) {
    super("User Input");
    super(parent, "UserInput", true);
    sourceStr = new String();
    destnStr = new String();
    requestStr = new String();
    setLayout(new BorderLayout());
    Panel panel0 = new Panel();
    panel0.setLayout(new GndLayout(6, 1, 0, 7));
    Label sourceLabel = new Label("Source Node ", Label.LEFT);
    source = new TextField("", 25);
    source.setEditable(false);
sourceList = new List(); sourceList.addItemListener(this);
destList = new List(); destList.addItemListener(this);

String choiceStr;
while (choices.hasMoreElements()) {
    choiceStr = (String) choices.nextElement();
    sourceList.addItem(new String(choiceStr));
    destList.addItem(new String(choiceStr));
}
choiceStr = null;

Panel panel1 = new Panel();
panel1.setLayout(new BorderLayout(0, 15));
panel1.add(sourceLabel, "West");
panel1.add(source, "Center");
panel1.add(sourceList, "East");
panel0.add(panel1);

Label destLabel = new Label("Destination Node", Label.LEFT);
destination = new TextField(" ", 25);
destination.setEditable(false);
Panel panel2 = new Panel();
panel2.setLayout(new BorderLayout(0, 15));
panel2.add("West", destLabel);
panel2.add("Center", destination);
panel2.add("East", destList);
panel0.add(panel2);

requestGroup = new CheckboxGroup();
allChkbox = new Checkbox("All Paths", false, requestGroup);
allChkbox.addItemListener(this);
shortChkbox = new Checkbox("Shortest Path", false, requestGroup);
shortChkbox.addItemListener(this);
exitChkbox = new Checkbox("Exit", true, requestGroup);
exitChkbox.addItemListener(this);

panel0.add(allChkbox);
panel0.add(shortChkbox);
panel0.add(exitChkbox);

okButton = new Button("OK");
okButton.addActionListener(this);
Panel confirm = new Panel();
confirm.setLayout(new BorderLayout());
confirm.add("East", okButton);
panel0.add(confirm);

add("Center", panel0);

setSize(350, 200);

/**
 * Processing for button events.
 *
 * @param e ActionEvent sent by the system.
 */
```java
/*
public void actionPerformed(ActionEvent e) {
    if(e.getSource() == okButton) {

        sourceStr = (source.getText()).trim();
        destnStr = (destination.getText()).trim();

        if(requestGroup.getSelectedCheckbox().equals(allChkbox))
            requestStr = "AllPaths";
        else if(requestGroup.getSelectedCheckbox().equals(shortChkbox))
            requestStr = "ShortestPath";
        else if(requestGroup.getSelectedCheckbox().equals(exitChkbox))
            requestStr = "Exit";
        else
            requestStr = "Exit";

        setVisible(false);
    }
}

/**
 * Processing for ListItem events.
 * @param evn ItemEvent sent by the system.
 */
public void itemStateChanged(ItemEvent evn) {
    String srcStr;
    String desStr,

    if(evn.getItemSelectable().equals(sourceList)) {
        srcStr = sourceList.getSelectedItem();
        source.setText(srcStr);
    }

    if(evn.getItemSelectable().equals(destList)) {
        desStr = destList.getSelectedItem();
        destination.setText(desStr);
    }
}

/**
 * Returns the chosen source string.
 * @return a copy of the source string.
 */
public String getSource() { return new String(sourceStr); }

/**
 * Returns the chosen destination string.
 * @return a copy of the destination string.
 */
public String getDestination() { return new String(destnStr); }
```
/**
* Returns the chosen request in string form.
*/
public String getRequestStr() { return new String(requestStr); }

/**
* Test method to display dialog box.
* @param args command line arguments; none needed.
*/
public static void main(String[] args) {
    Frame theFrame = new Frame();
    Vector list = new Vector();
    UserlnputGUI u = new UserInputGUI(theFrame, list.elements());
    u.show();
    System.exit(0);
}

/**
* A class representing a Base class for all user requests.
* @author Stephanie Burton
*/
public class Request{

/**
* Creates a Request object.
*/
Request(){

/**
* Performs the behavior of the request.
*/
void doRequest(){
}
import java.util.Vector;

public class AllPathsRequest extends Request{

    private String source;
    private String destination;
    private Graph theGraph;

    AllPathsRequest(String src, String destn, Graph grph){
        source = src;
        destination = destn;
        theGraph = grph;
    }

    public void doRequest(){
        Vector scratch = new Vector();

        System.out.println(" ");
        System.out.println("All Paths from "+ source + " to "+ destination + ": ");
    }
}
theGraph.allPaths(source, destination, scratch);
System.out.println(" ");
}

import java.util.Vector;

/**
 * A class for collecting pertinent information to find the shortest
 * path from a source to a destination node in a specified graph.
 * @author Stephanie Burton
 */
public class ShortestPathRequest extends Request{
  /**
   * String name of the source node.
   */
  private String source;

  /**
   * String name of the destination node.
   */
  private String destination;

  /**
   * Specified graph to perform the request on.
   */
  private Graph theGraph;

  /**
   * Creates a ShortestPathRequest object
   *
   * @param src the source node name.
   * @param destn the destination node name.
   * @param grph the graph to search.
   */
  ShortestPathRequest(String src, String destn, Graph grph) {
    source = src;
    destination = destn;
    theGraph = grph;
  }

  /**
   * Searches the specified graph for the shortest path from
   * the source node to the destination node.
   */
  public void doRequest() {
// View shortest path
System.out.println(" ");
System.out.println("Shortest path from " + source + " to " + destination + ": ");
Vector path = theGraph.shortestPath(source, destination);
if(path.size() == 0){
    System.out.println("Destination could not be reached");
    System.out.println(" from the source");
}
else{
    System.out.println("Path Size = " + path.size());
    System.out.println(source);
    for(int i = 0; i<path.size(); i++)
        System.out.println(" " + ((GraphNode)path.elementAt(i)).getName());
    System.out.println(" ");
}
System.out.println(" ");

/**************************************************************************************************
// File: ExitRequest.java
// Purpose: class performing a request to exit application.
/**************************************************************************************************

/**
 * A class representing a request to exit the application.
 *
 * @author Stephanie Burton
 */
public class ExitRequest extends Request{

    /**
     * Creates an ExitRequest object.
     */
    public ExitRequest() {} 

    /**
     * Performs a system exit.
     */
    public void doRequest(){
        System.exit(0);
    }
}