Design Patterns by Example

Garrett Mitchener

July 21, 1997

Contents

Chapter 1

Introduction

1.1 Our job as programmers

So, you want to write a program? Assuming you've already figured out what it's supposed to do (and that subject alone warrants its own book), the next step is to plan the program and figure out how to do all that stuff you promised, then get on to writing it.

Object oriented programming is a collection of techniques intended to make our job as programmers easier. In particular, it's supposed to save us time and effort. Under the best of circumstances, the hard part of a task is written once, and then we can simply adapt it to our needs later. If we suddenly come up with something neat to add to our program, it should be easy to add in, and not require the use of a sledge hammer or its silicon equivalent.

However, these simple concepts of reusability and maintainability also makes our job very difficult: Not only do we have to get it done, we have to get it done "right," meaning in such a way as that we can actually understand later how it works and fix or adapt it.

So, we actually end up with two jobs. The first is as an application programmer. In this guise, we have to write some code which is likely to be totally useless in any other program. Command line parsing, for example, is not completely reusable unless you happen to have two programs which take exactly the same options. Help messages and how the menus are organized are also not particularly reusable. They must still be done "right," so we can come in later and change things, but that part of the task isn't usually too hard.

The second job is as a library programmer. This is more difficult, because code that goes into a library should be as reusable as possible. Any application-specific operations must slide seamlessly into the framework without requiring that anyone edit our code in the future. Since we as library programmers can't predict everything our library will be used for, the best we can do is shoot for some sort of general, powerful solution. As we write more programs, especially related ones, the libraries from all of them should build on one another, and in the end, the amount of thrown-away code put into the applications themselves should be minimal.

Object oriented programming has turned out to be a very successful means of maximizing the library and minimizing the thrown-away code.

1.2 Tools of object oriented programming

The techniques of object oriented programming make use of several layers of thinking. At the bottom are simple things, such as classes, objects, functions, and variables, that make up the building blocks. There are three higher level tools available in object oriented programming: encapsulation, intelligent data, and inheritance.

1.2.1 Encapsulation

Encapsulation is the so-called "black box" data model. All objects are like machines with a control panel on top, an instruction book, and a label on the access panel that says, "Do not remove under penalty of law."

The premise here is that as application programmers, we need to know what objects are capable of and how to get them to do it, but not how it actually works. This saves us time, since there is no need to figure out how someone else's code works. It also allows us to utilize someone else's library without depending on how its innards are written. When it's changed or updated or ported to a new operating system, the differences are invisible from the application's point of view, hidden inside the black box, so our code can simply be re-compiled and should still work.

Encapsulation also prevents name conflicts. In other words, if everything a class needs is a member of that class, then there is less chance that someone else's code will conflict with it by defining a different variable or function with the same name. For an example of what not to do: The GNU implementation of the Standard Template Library defines symbols "red" and "black" at global scope (instead of inside the balanced tree class that uses them) and is not usable with a GUI toolkit called Qt that defines a bunch of colors at global scope.

Encapsulation also makes it easier to design and plan programs. Just about every large program appears unmanageable at first. Since object oriented programming encourages us to package functionality into classes, planning a program boils down to deciding which operations should be grouped together, and how the classes, rather than the individual functions, should interact.

1.2.2 Intelligent data

Object oriented languages allow functions to be attached to objects. They are therefore intelligent and carry their functionality with them. In C++, this is accomplished through virtual functions. This principle is at the heart of many of the most powerful design patterns, especially those which carry custom operations into an already existing framework. The technical name for this is "polymorphism," meaning "many forms."

1.2.3 Inheritance

Inheritance means the process of adding new operations and data to an old class. There are several somewhat different uses for inheritance.

Programmers can use it to take advantage of code which is already written, a pattern known as a Template Method. It's common for a library to provide an abstract superclass where the "hard part" of the programming has already been done. A derived class must simply provide a few specific functions which fit in with the parent class's. For example, the `ResourceUser` class described in section 4.5 provides correct implementations of tricky and error prone operations which allow it to make a limited resource appear unlimited. Derived classes has to implement "primitive" functionality which actually acquires and releases the resource.

‡*Template Method, 325*

In a strongly typed language, inheritance becomes important as a means of making data intelligent. The compiler won't let us call a method on an object unless we tell it ahead of time which operations we need to be able to use on that object. We can write a purely abstract class which defines an interface. Any code written in terms of the abstract calss can call functions in that interface and will work on any object that inherits from the abstract base class. This is how intelligent data works in C++.

1.3 Design patterns

Design patterns are the next level up in our toolbox. They build on top of the lower level tools and provide a way of dealing with large scale parts of the design of a program.

In particular, the program as a whole breaks down into smaller problems: What's the best way to create this complex object? How do we encapsulate this action? How do we allow for multiple look-and-feel standards?

Many of these problems can be solved with well planned interactions between classes. The general form of the solution is the design pattern. One of the goals of object oriented programming is to write reusable code. Design patterns are a sort of reusable thinking.

The rest of our discussion will focus on the patterns cataloged in the book *Design Patterns* [2]. In particular, we will look at a specific program and how to apply those patterns to its design. The little boxes in the margins refer to pages in the book where certain patterns are described in detail. (Think of them as hyperlinks.)

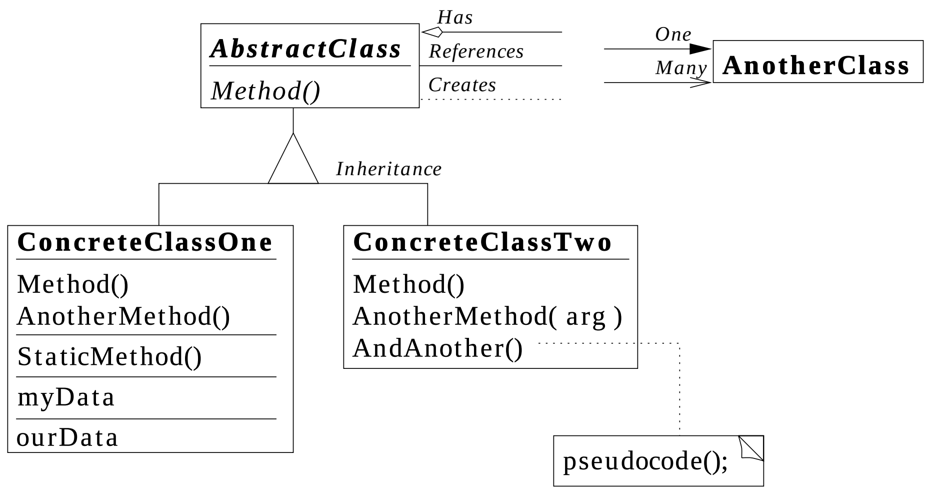

Figure 1.1: Sample class diagram.

1.4 Design notation

Since class and object interactions can be difficult to discuss verbally, a number of visual methodologies have been invented. The one used here is based on the notation in *Design Patterns*, a variation of OMT notation. See figure 1.1 for an example.

Classes are represented by boxes. Their names are at the top followed by member functions, static member functions, instance data, and static data[1]. Not all sections are present in all class boxes. Instance data members all have names beginning with "my," and static data members all have names beginning with "our." Return types and types of variables are usually left out of the diagram to make them less confusing. Functions are almost always public, and data members are always private. Italic text indicates that a class or function is abstract, while upright text indicates a concrete class or function.

Subclasses are connected to their parent class by a tree of lines with a triangle at the trunk.

Class interactions are denoted by lines and arrows. A solid line with a diamond at the bottom indicates that one class contains instances of another. A plain solid line means that one class uses another. Dotted lines show that a class creates instances of another. A solid arrow indicates that just one object is being used, contained, or created, while a forked arrow means that many objects are involved. Remember that the base of the line is attached to the container, user, or creator, and the arrow end is attached to the class being contained, used, or created. Not all relationships are shown in every diagram.

A box with a dog-eared corner contains a pseudocode implementation of a

[1] "Static" is C++ terminology for functions and data which belong to a class as a whole rather than to an individual instance.

function. These are used to give a general idea of how a class actually works.

Chapter 2

The example

2.1 The original assignment

The example program used here to illustrate design patterns is called Oodle, the **O**bject **O**riented **D**irectory **L**isting and **E**xpansion program. It was assigned to a CPS 108 class in the spring semester of 1997 and consists of two big parts.

The first or "interactive" part allows users to view a list of all the files in a directory in order by name, size, or modification date. Additionally, they can navigate the file system and view different directories.

The original assignment also required each programming group to write an object oriented replacement for the BSD function `scandir()`:

```
#include <dirent.h>

int scandir(const char *dir, struct dirent ***namelist,
            int (*select)(const struct dirent *),
            int (*compare)(const struct dirent **,
                           const struct dirent **));
```

This function reads the contents of a directory given its name, picks out only those files for which `select` returns true, and sorts them according to `compare`. The results of all this are stored in `namelist`. This function is quite general, saves us a lot of work, and is a pretty good overall solution to the problem of scanning and sorting files in a directory. Unfortunately, it uses dumb, low-level data structures, and calls `malloc()` rather than `new` to allocate memory for the array of file names. Also, it is implemented only on BSD UNIX systems. So, an object oriented replacement is called for.

The second or "comparison" part of Oodle has two modes. In log mode, the program recursively traverses a directory tree, saving information about the files in some more or less permanent fashion. In diff mode, the program recusrively traverses the directory tree and compares it to the logged version, displaying any changes it finds. As an extra detail, the output of diff mode must be "pruned:"

If a directory has been removed since the log was made, only that directory must be mentioned. We know all of its contents have been deleted. Similarly, if a new directory is found, we know all of its contents are new as well, so only the new directory may be mentioned. For brevity, these two operations will be referred to as "logging" and "diffing."

We're going to discuss **`scandir()`**, logging and diffing in detail. In particular, we're going to look at a number of solutions turned in by students in the class and how to use design patterns to improve the design. The user interface is left for later because to do it right would require designing a terminal widget kit which I don't have time to do yet. . . .

***

Chapter 3

Before

3.1 Rules of the game

We're going to play "Before-and-After" now.

First, we'll look at some of the programs turned in by students who took the class in some detail, looking carefully at places where the design can be improved with some patterns. The Oodle assignment was the first one given, so most of these designs were assembled without the benefit of experience. They represent each group's first attempt at using object oriented techniques to implement a large-scale program. Most of the students had just taken a course on data structures and were familiar with the concept of encapsulation, but intelligent data and inheritence were new to them. Currently, functional programming is still taught in introductory courses, and object oriented techniques are not taught until later. There are therefore some traces of functional mind-set in these designs.

None of these designs are "bad" per se, but there are ways to improve all of them. The idea is to maximize the reuseable library part, and minimize the throw-away application part.

You might want to take some time now before reading on and try to design Oodle for yourself. Don't cheat and look ahead, or we'll have to give you fifty lashes with the scrabula. Once you have your design, pay careful attention to the "Before" part and see what choices you've made that parallel the examples given here. Not all of those choices will be "bad," but some of them will clearly represent non-object oriented ways of thinking that you should probably reconsider. Additionally, think about what exactly constitutes a design, and which parts of your code will be library-worthy, and which parts are application specific.

One last thing: The names have been made up to protect the innocent, but the designs are real....

3.2 Before

3.2.1 Data structures

All the students taking CPS 108 had access to a selection of fundamental data structures provided by a library called Tapestry. See figure 3.1 and [1]. Many of the data structures have a `MakeIterator()` method which creates an instance of a companion iterator class that passes over each element of the data structure. Whatever receives the iterator must be sure to delete it. Additionally, there is an `IterProxy` class which stores and gives access to a pointer to an `Iterator` object. When the proxy goes out of scope, it deletes its iterator. This is just an easy way to prevent a memory leak. Additionally, a fairly powerful string class is available, as are classes for reading the contents of a directory without resorting to system calls. In the design diagrams that follow, fundamental data structures have been left out to save space and confusion.

‡*Iterator, 257*

‡*Proxy, 207*

Design by Boar, Land and Associates

For the class diagram, see figure 3.2.

State information about non-container files is contained in `FSItemInfo`. Its `GetContents()` function does nothing. Since a directory is a special type of file, a directory could be represented with a `FSItemInfo` object. The `IsDirectory()` method returns true if this is the case.

Directories are more completely represented by class `DirInfo`. Internally, it represents its contents with a vector of `FSItemInfo` pointers. `GetContents()` returns this vector. The `GetFileInfo()` function sequentially searches its contents for a file of the given name and returns the pointer it finds. `GetDirInfo()` works similarly, but examines only directories. When a `DirInfo` is created, it recursively creates objects for all of its contents. This has the side effect that it takes a long time to create a `DirInfo` if the directory tree within it contains a large number of files. As an interesting side note, the writers of this project did not make the `FSItemInfo` destructor virtual and ended up jumping through some hoops to get rid of the resulting memory leak.

The `SortFacade` class contains three vectors of pointers, each of which is sorted in one of the three required orders. As each sort order is requested, the `SortFacade` fills in the vector once and sets a flag indicating that it has been sorted. That way, sorting for each order is done only once. There is no way to add another sort order.

Class `Comparer` is responsbile for printing out differences between the log and the current state of the file system. It has a hash table called `myMap` from file path names to `FSItemInfo` pointers, which represents the entire current state of the file system. The public function `PrintReport()` does just what its name suggests: print all differences betweeen the log and the current state of things. It calls a number of private functions to facilitate this. `LoadMyMap()` traverses the file system tree using `DirInfo` objects, and stores all the information in `myMap`. The private function `Compare()` reads the contents of the log file

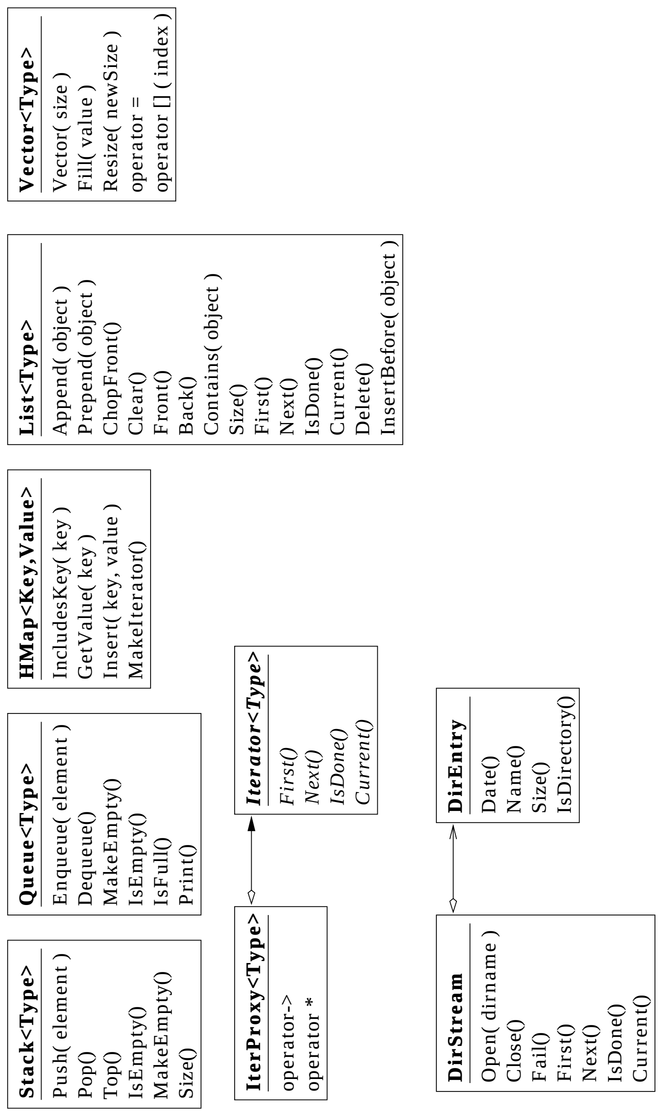

Figure 3.1: Data structures available to the student programmers in the Tapestry library.

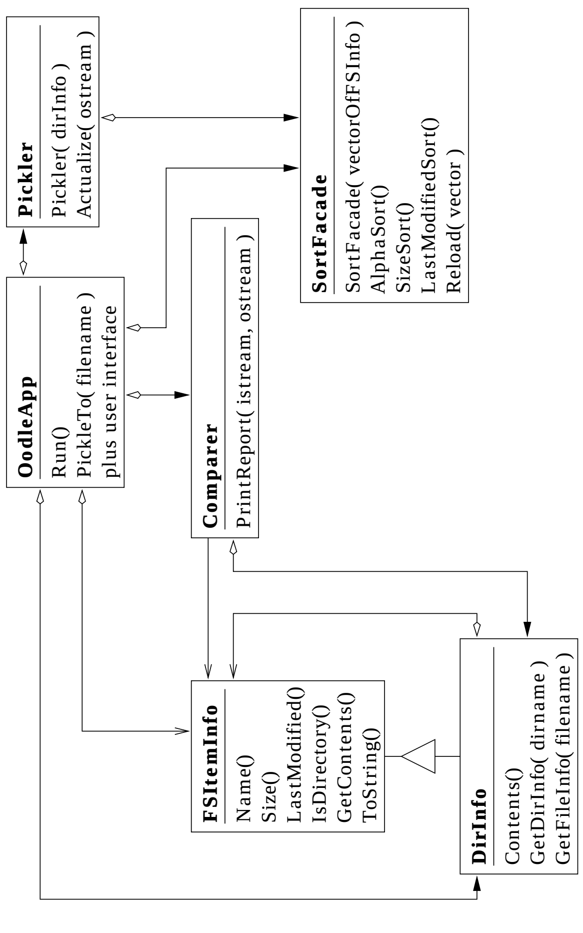

Figure 3.2: Design diagram by Boar, Land & Associates.

sequentially, looking for deleted and changed files. It writes a short message for each changed or deleted file. The output of diffing must be pruned as described in the assignment. To implement this, the contents of any deleted directory are placed in a vector. Before any file is displayed, this vector is searched sequentially, and if the file is found, it's not displayed. `CheckForNew()` uses another traversal of the physical file system to check for newly created files.

Logging is performed by class `Pickler`. (The name comes from the concept of pickling food for long term storage. We can do the same with files.) Its `Actualize()` method uses a queue to traverse a file system using `DirInfo` objects. Each line in the log represents a single file. The full path to each file is stored, and the log file is a flat list structure, not a tree.

Class `OodleApp` is the central application class. It contains the user interface and directs what the prgram should do next. It contains code to display a help screen, list files in the three required orders, and parse user input. It uses a `Pickler` to do logging, and a `Comparer` to do diffing. `main()` creates an `OodleApp` and sends it high-level instructions based on the command line.

What's funny about this design is that the class structure has almost nothing to do with the abstractions used by the program. Upon inspection of their source code, it becomes clear that what we actually have is four different representations of a file system.

File tree: A representation of a file system as a tree. Leaf nodes are represented by `FSItemInfo` and nodes with children by `DirInfo`. This tree is capable of traversing itself, printing changes relative to a file map, and writing itself to a file list.

File map: A representation of a file system as a mapping from file names to information about them. These maps can be created from a file tree.

File list: A sequential list of files based on an `ostream` which can be read once from beginning to end.

File vector: A sequential list of files in an array sorted in a particular order.

These are the key abstractions used in the program, despite the fact that their functionality is strewn all over a handful of unrelated classes. There is also some confusion over who owns what, since most of the time, the map representation refers to objects which appear to belong to a tree representation. In all the chaos, we end up with a data structure that looks like a dictionary, smells like a tree, and barks like a chicken.

The only use of an object oriented technique in this design is in the tree form of the file system, which is slightly polymorphic.

The user interface is okay, but implemented as a tangle of `if-else` chains. Error messages are put in all over the place, so if we wanted to write a different interface or translate it into French, we would have to go in and manually change zillions of `cout <<` statements in all the different classes.

The design includes no replacement for the `scandir` function. There is also a problem with the recursion. Under UNIX, a program can only have a small number of file handles open at once, and reading a directory uses one. If directory recursion goes too deeply, this program runs out of file handles and crashes.

Design by Microsquish, Inc.

For the class diagram, see figure 3.3.

To begin with, this particular Oodle has a well-designed user interface which isn't shown in the class diagram. It uses a sohpisticated menu class and the Command pattern which effectively decouples the `Application` class from most of the others.

‡*Command, 233*

Individual files are represented by objects of class `ScanDirEntry`. In addition to providing state information, the class includes methods for reading from and writing to streams.

The `ScanDir` class and its companion `ScanDirEntry` replace the `scandir()` function quite well. `ScanDir` represents a directory and behaves like a shallow container. The `MakeIterator()` function creates a `ScanDirIterator` which makes the contents available. Unfortunately, the iterator class doesn't inherit from an abstract base class, which would make it more useful. This is an example of the Iterator design pattern. Additionally, a class called `ScanDirIterProxy` is included which contains and provides access to a pointer to a `ScanDirIterator`. When a proxy object goes out of scope, it automatically deletes the iterator. This is useful because it prevents a memory leak that would occur if we asked a `ScanDir` to create an iterator for us, then forgot to delete it at the end of the function. This is an example of the Proxy design pattern. `ScanDir` objects are capable of writing themselves to `ostreams`.

‡*Iterator, 257*

‡*Proxy, 207*

`PickleJar` is an interesting class. It's a sort of persistent hash table from file path names to their state information. The program uses this class to read and store the old state of the file system. Internally, it uses a hidden directory and several specially named files to store the information. `PickleJarIndex` is an auxilliary class which determines which file to fetch information from based on a path name.

Apparently, `Pickle` was supposed to be a base class for objects which could store themselves to a file, but it isn't used at all in the rest of the program, and the storing and retrieving code is all in terms of `ScanDirs`. The four member functions (according to comments in the code) are supposed to do the following things:

- `Preserve()` should copy the current state of the object into temporary storage.
- `LogPickle( ostream )` should write the stored state to the stream.
- `RetrievePickle( istream )` should read a stored state from the stream into temporary storage.
- `Restore()` should move the state information from temorary storage into the object's accessible data.

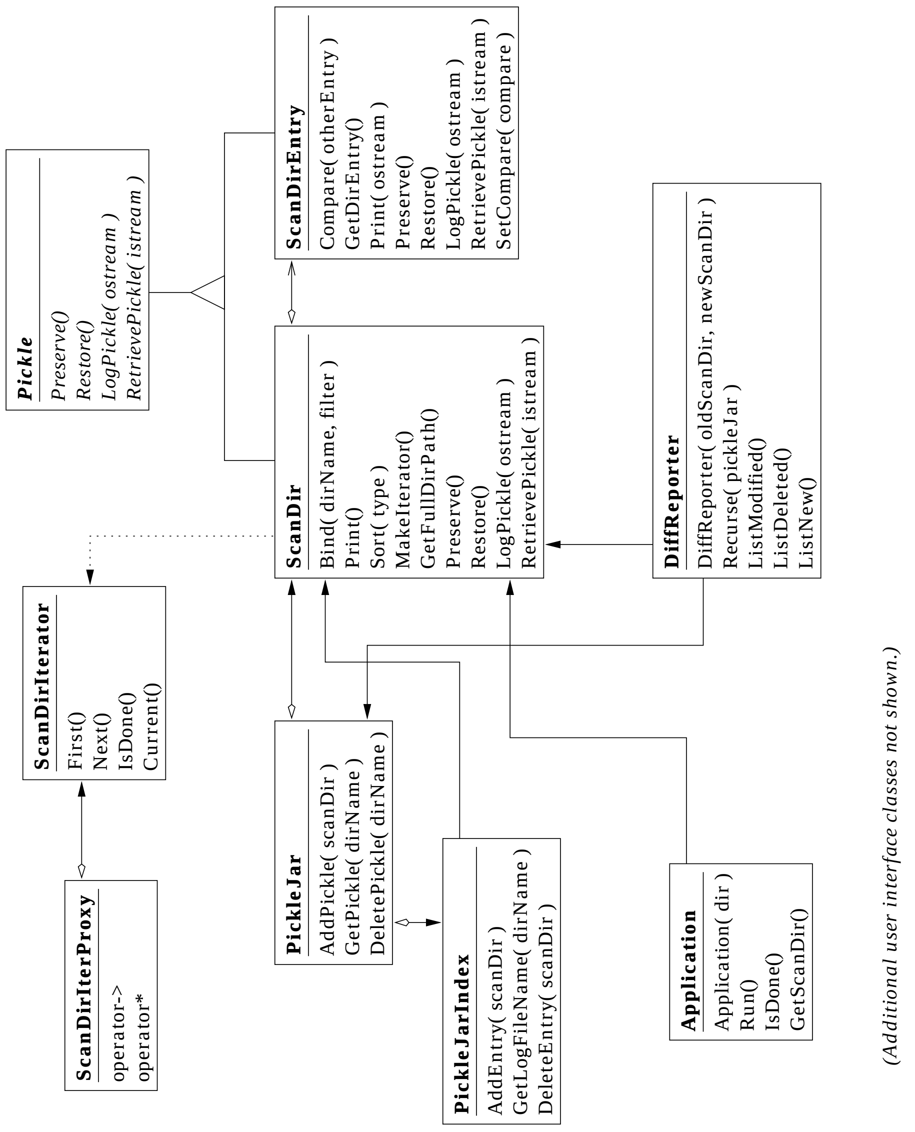

Figure 3.3: Design diagram by Microsquish, Inc.

Class `DiffReporter` is responsible for reading in the old state of the file system and comparing it to the current state. It non-recursively slurps the contents of the old `ScanDir` and the new one into two hash tables from path names to `ScanDirEntries` for easy access. The `Recurse()` function does most of the work. It uses a `PickleJar` to retrieve stored entries. In each directory, it prints out the new, deleted, and changed files, then creates another `DiffReporter` for each subdirectory and has it print out its report, and so on.

This design is pretty good, all things considered. Since most of the user interface is separate from the computation, it's easy to change the interface. Unfortunately, the `DiffReporter` class does its own output, so if we wanted to do something with the changed files other than just list them, we'd have to either copy the existing code and change it, or start from scratch. Also, the `PickleJar` class, while it appears to be an attempt at providing a generalized persistent hash table class, is mostly useless outside the context of Oodle. Once again, we have two different ways of representing a file system: one as a series of nested containers (`ScanDir` objects) and again as a dictionary (`PickleJar`).

There is one strange thing about the sorting system. Class `ScanDirEntry` contains a static member which is a pointer to a comparison function. All sorting takes place in terms of that pointer's contents. A more common and possibly better technique is to pass the comparison function to the sorting algorithm.

Design by Superego Software

See figure 3.4 for the class diagram.

The `ScanDir` class is something like an array of files which represents a directory. The `List()` function returns the name of a different contained file each time you call it. Each file is associated with a number, and the `GetIndexName()`, `GetIndexTime()`, and `IsIndexDir()` return information about a file given its number. The `GetLength()` method returns the number of files in the `ScanDir`. `SetFilter()` takes a pointer to a filter function which should return true if the `DirEntry` passed to it should be included in the list. A directory is read in with the `ReadDirectory()` method, which only reads in the files accepted by the filter. Sorting is done with two functions. `SortEntries()` takes an integer flag, which selects either by name, by date, or by size. `CustomSort()` takes a function (*not* a comparison function) and applies it to its internal vector of `DirEntries`. Apparently, the passed in function should sort the vector, although it could just as easily do anything else. `DiffScanDir()` compares the contents of the object to the information stored in an input stream, stores string representations of files which have changed in the passed in vector `diffs`, and puts names of subdirectories in the supplied vector `subdirs`. `FormatLine()` takes a `DirEntry` and returns a string representation of it suitable for display on a screen 80 characters wide.

Class `Oodle` has a hash table of path names to `ScanDir` objects which it collaborates closely with. A single `Oodle` object is created by `main()` and sort of runs the show by communicating with the user interface. The program maintains a hidden directory where it stores log files. `CheckLogFiles()` is called

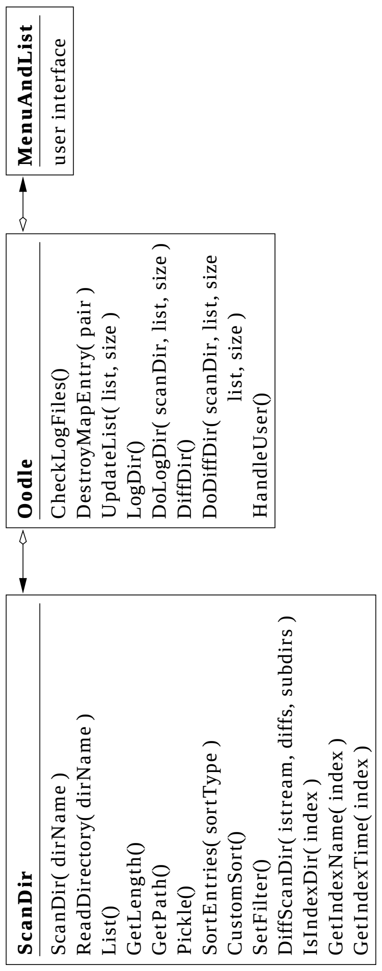

Figure 3.4: Design diagram by Superego Software.

upon start-up and looks in the hidden directory for invalid log files or logs which are very old. One `ScanDir` is maintained for the current directory and `UpdateList()` puts a string representation of each of its files into passed-in vector. The `LogDir()` method simply calls the recursive `DoLogDir()` which stores information about `ScanDir` objects in specially named files in the hidden directory. Likewise, `DiffDir()` calls the recursive `DoDiffDir()` which compares log files to `ScanDirs` created from the physical file system and fills a vector with string representations of any differences it encounters. `HandleUser()` makes calls to the interface class to interact with the user. A few other functions are provided for moving the `Oodle` to a new current directory.

This design does show good separation of user interface from computation. The interface class knows nothing about `ScanDirs` or `Oodles` and stands a good chance of being reusable. The convention of passing vectors of string representations around makes this separation possible, although there are probably better ways to do it.

The `ScanDir` class does indeed replace the BSD `scandir()` function, but not particularly well, since it's quite awkward to get the sorted and fitered information out of the object. Also, that `CustomSort()` could use a little work since it saves the programmer exactly zero work.

Unfortunately, the `ScanDir` and `Oodle` classes are a case of the "god class," in which one object ends up doing a little too much. The `ScanDir` class has to do iteration, several different kinds of output formatting, parsing saved input, sorting, filtering, diffing, and logging. `Oodle` has to do some log retrieval, part of the diffing, more formatting, and drive the user interface. The two classes provide functionality in a lot of the same areas, such as how the log directory works, and could easily step on each other's toes if we wanted to change the program around.

Again, the primary abstractions in this program have nothing to do with the classes. They are:

Directory: A sortable, filterable list of file information, capable of comparing itself to a saved copy.

File: Information about a single file. (For whatever reason, the `ScanDir` class returns only bits and pieces of a file by its number. This is reminiscent of the technique of having parallel arrays of differeny types, used frequently in old-fashioned BASIC, which had nothing like a `struct`. Furthermore, these functions are hardly used at all by the program, so it is unclear why they are even there.)

Persistent file system: A permanent hash table of some kind from path names to information about the files.

Functions and data for these tasks are split between `Oodle` and `ScanDir`.

Other difficulties of note

Those three are typical of the designs turned in for this assignment. Some were better, others were worse. Some worked, some didn't. Some had good interfaces, some didn't, but that's another story and shall be told another time.

Concerning designs, here is a list of many of the mistakes, bad ideas, and things-to-be-improved-upon that showed up frequently.

Mistaking a data structure for an abstraction. File systems have to support an operation where we give it a path name and it gives us back information about that file. Although that does sound like a job for a hash table, the file system is really tree shaped, and the fact that paths are represented by strings is just tradition, so a tree-like data structure really makes more sense. Many groups just used a plain `HMap` object, without even wrapping it up in a separate class. Then, when they needed to attach more functionality to the file system, such as recursion, it had to go elsewhere, and led to chaos in the design.

Using nondescriptive names. Many groups had an `Oodle` class, or something like that. But what is an `Oodle`? Or for that matter, what is a `ScanDir`? Other wonderful names used by various groups included `myMap`, `myVec`, `myQ`, `stupid`, `stoopid`, and `stooopid` (no joke!). The idea here is that if you don't know what to call it, then you don't know what it really is. If you can't figure out what it's name means, then you don't know what it really is and neither did whoever named it. If you don't know what it really is, you can't code it and have it make any sense at all.

Crossed wires. A consequence of not knowing what our abstractions are is that we end up putting member functions and data for them in several different places rather than in one class. Along with this comes confusion over who owns what. If we have an `HMap` of path names to pointers to file information, and a vector of pointers to the same file information in alphabetical order, then which object is responsible for deleting what? Crossing of wires is often a consequence of using nondescriptive names or mistaking a data structure for an abstraction.

Doing things the hard way. It's amazing how many groups provided completely different and unrelated mechanisms for doing the three sort orders required for the user interface and the custom sort order required for `scandir()`. Often, there was a global flag which could be set to "name," "date," "size," or "custom," and the sort function (wherever it was) would use a `switch` statement on that flag to determine which one of four helper functions to call. Why not just solve the general problem, since we have to solve it anyway, and implement solutions to the specific problems in terms of the more general solution? (A few groups did in fact do this.)

Mixing user interface with computation. The Oodle assignment was given before any mention was made of GUI's, and it was assumed by just about everyone that it would always run in a console (text-only) mode. As a result, many of the programs have output statements in the diffing code, report all errors to `cout`, and have menus in random places.

It's a bad idea to tie computation directly to user interface. What if we decide later to port the code to a windowing interface? What if someone wants to use it in a minature electronic memo book which has a one line screen? The user interface of a program is very seldom reusable[1], so we should make every effort to keep it separate.

Concerning errors: It's okay for a program to handle logical errors (things which should *never* happen) in a non-graceful manner, such as by printout out a desperate message and exiting. In the case of user input errors however, the program should pass the information along to the user interface, informing the user that something they did won't work and asking for what to do next. Since this involves communicating with the user, it should be handled by the user interface, and therefore kept separate.

Very little use of object oriented tools. Most of the programs reeked of functional programming. Very few of the designs used inheritance, and even fewer used it in a way that made any sense. Virtual functions were few and far between, and groups tended to think in terms of data structures and implementations rather than abstractions. Even encapsulation tended to be violated.

A few of the groups did in fact use the Command pattern in their user interfaces, and benefitted greatly. The Iterator pattern showed up frequently, too. Class discussion encouraged the use of these two patterns.

‡*Command, 233*

‡*Iterator, 257*

[1]Although small components of it, such as buttons and lists, often are.

Chapter 4

After

4.1 A more elaborate design

This chapter is devoted to a detailed description of a completely different design which makes extensive use of design patterns. In the first few sections, we will discuss the extensive library code. The library is called "Bargello," after a style of needlework made famous by the Bargello museum in Florence, Italy.[1] The small amount of Oodle specific code is put off until the end to emphesize the point that with the Bargello library in place, Oodle itself is almost trivial.

The various frameworks within Bargello are described roughly in order of increasing complexity.

Don't get the wrong idea about complexity. Part of object oriented design involves trading one form of complexity for another. In the "Before" section, one of the designs had only three classes but was arguably the most difficult to understand. The others are hard to figure out because the interactions between them are complex and unclear. Code which uses chains of `if-else` statements or `switch` statements is structurally simpler than, say, polymorphism, but much harder to read and comprehend.

Object oriented deisn trades all that for a bunch of smaller, intelligent classes, with specific interactions. Keep that in mind as you look at figure 4.1 which shows in minature most of the 56 classes present in Bargello.

4.2 Magic pointers

4.2.1 Intent

‡*Proxy, 207*

To provide a general purpose Proxy for pointers.

[1]The style makes use of abstract patterns.... I know it's a bad pun, but it's easy to spell and goes with Tapestry.

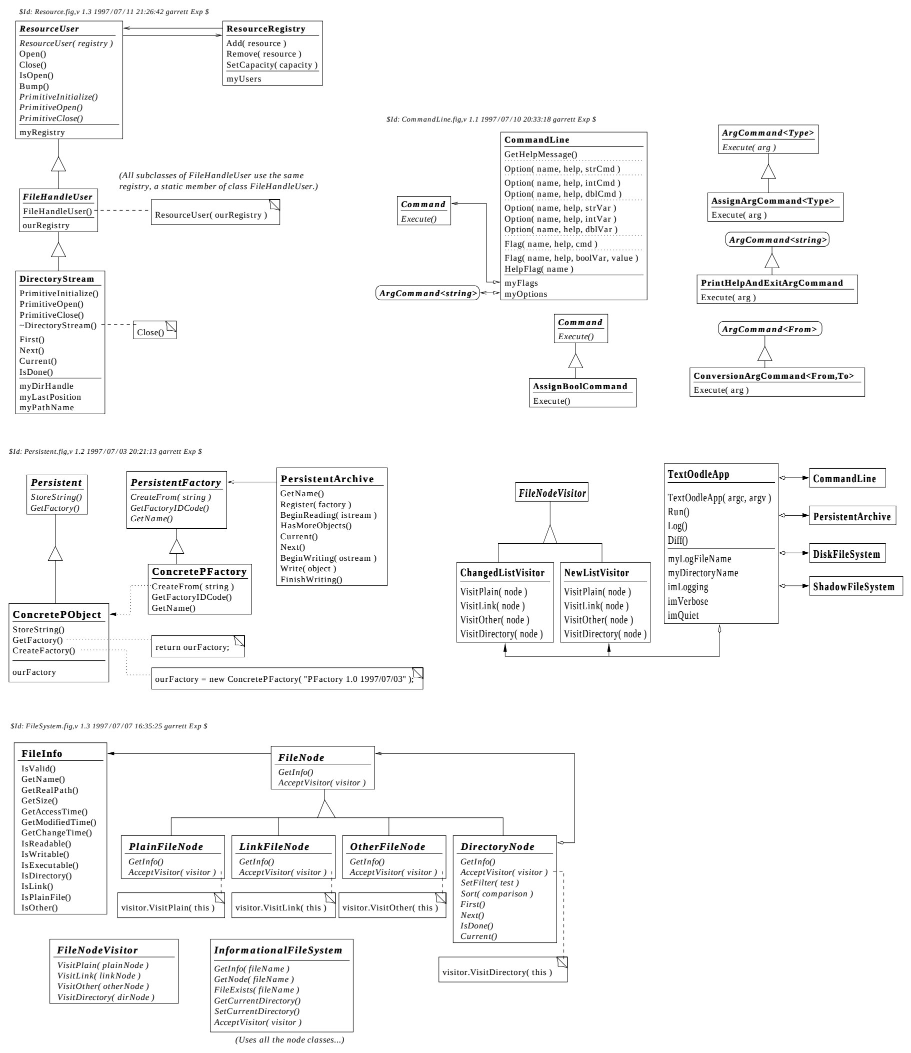

Figure 4.1: All of Bargello and Oodle.

4.2.2 Motivation

Throughout the Bargello library, a number of complex creational patterns are used and as a result, it is very easy to create a memory leak. For example, the file system classes have a “create node” method which creates a new object and returns a pointer to it. When should that object be deleted? It could be stored as part of another object, so it should be deleted when that object goes out of scope, or used temporarily, in which case it should be deleted as soon as possible.

The problem boils down to a question of ownership: When does a pointer own it’s contents? If it does, then it should delete it when it goes out of scope.

4.2.3 Solution

The `MagicPointer` class is one solution. Each instance of it contains a pointer and a flag indicating whether or not the object is responsible for deleting its contents (the “pointee”) when it goes out of scope.

```
template<class Type>
class MagicPointer
{
public:
    MagicPointer( Type * pointee = 0, bool owns = true );
    MagicPointer( MagicPointer<Type> & );

    ~MagicPointer( void );
    void Destroy( void );

    MagicPointer<Type> & operator=( MagicPointer<Type> & );
    MagicPointer<Type> & operator=( Type * pointee );

    Type & operator*( void ) const;
    Type * operator->( void ) const;

    MagicPointer<Type> & SetOwnership( bool flag );
    MagicPointer<Type> & PointTo( Type * pointee,
  bool owns = true );

    bool Owns( void ) const;
    Type * Pointee( void ) const;
    bool IsNull( void ) const;
};
```

Instances of this class look very much like pointers thanks to the overloaded `*` and `->` operators. The magic part comes from the fact that when a `MagicPointer` goes out of scope, it first checks to see if it owns its pointee. If it does, it deletes it. Since this class represents a substitute for dumb pointers, it is an instance of the Proxy design pattern.

‡*Proxy, 207*

This has several advantages. For instance, if we have a class that must store something by pointer, we can use a `MagicPointer` for the data member instead of a dumb pointer and a simple member-wise destructor supplied by the compiler takes care of deleting it automatically.

If we call `MakeIterator` on a data structure, we can store the returned pointer in a `MagicPointer` and it will be deleted automatically.

The tricky thing about `MagicPointers` is their copy semantics. When one is copied, ownership of the pointee is transferred to the copy. This is so that `MagicPointers` may be passed by value to other functions.

A word of warning: This class will not solve all your memory management problems. For example, if a `MagicPointer` is copied and the copy goes out of scope before the original does, then the original points into oblivion and your program will most likely crash if you try to use it.

A copy constructor and assignment operator are provided which work on `const` objects (not shown in listing). This is a necessary evil because many container classes require just such member functions. The `MagicPointer` copy semantics require that ownership be transferred to the copy so the implementations of these functions must cast away the `const` of their argument and call `SetOwnership()`. This action is usually harmless, but if you use this class, you should be aware of it.

This class is very similar to the `auto_ptr` class, which is probably going to be included in the Standard Template Library. For a good discussion of the smart pointer idiom in C++, see [3].

4.3 Encapsulating actions

4.3.1 Intent

To encapsulate actions and the means of passing them the information they need.

4.3.2 Motivation and reformulation

Many activities in a program are event-driven. That is, the program is supposed to perform some action when a particular condition arises. Parsing and interpreting user input are often event-driven, and so are network connections, simulations, and lots of other things.

There are two ways in general of coding an event-handling program. One is to write a loop that checks for each possible event and performs an action based on a chain of logic. The trouble is that objects which have to handle a lot of events get to be very difficult to code, and adding or moving event-handling code around becomes a maintenance nightmare.

A better way would be to encapsulate the action somehow and use a more intelligent means of storing and finding it, such as a hash table, or a Chain of Responsibility pattern. Often, just a direct reference will do, i.e. a button that knows what to do when pressed.

In particular, we want to be able to do the following things:

- Add an indefinite amount of functionality to an object.
- Reference any data the action needs without resorting to global variables or anything similar.
- Be able to pass parameters to the action.

4.3.3 Solutions from other languages and libraries

As a note to the reader, many of the code examples in this section contain abbreviations, such as shortened or slightly altered names, missing details, and occasionally omitted syntax. This is because the examples span a wide range of languages and libraries, and most readers will not be familiar with all of them. Rather than obscure the example with a lot of hard-to-explain details, I have sacrificed exact correctness in favor of clarity and consistency.

In many interpreted languages, it's possible to pass pieces of code around using what's known as a code block, bound method, closure, or callback, depending on what language we're using. Suppose for instance that we have a GUI toolkit and we want to cause our program to exit when someone presses the "quit" button. What we'd like to do in general is store some code in a variable somehow and have the button execute it when pressed. The program quitting operation is simply an example. So, suppose we have a `Button` class with a "when-pressed" method of some kind that takes a bit of code and stores it away, to be executed when the button is pressed. Suppose also that we've decided to write a subclass of `Button` called `QuitButton` which mostly consists of an initialization function that installs our little bit of code. We must also have an `Application` class which includes an `Exit()` member function.

Here's what it might look like in Python:

```
class QuitButton(Button):
    def initialize(self):
        self.setLabel( "Quit" )
        self.whenPressed( self.quit )
        return self

    def quit(self):
        self.application.exit()
```

or in Smalltalk:

```
Button subclass: #QuitButton
       instanceVariableNames: 'myApplication'
...

!QuitButton methodsFor: 'initialization'!

initialize: app
    myApplication := app.
    self setLabel: 'Quit'.
    self whenPressed: [myApplication exit].
    ^self

!!
```

Here's the same sort of thing in Perl:

```
use Button;
package QuitButton;

@ISA = ( 'Button' );

sub initialize {
  my ($self, $app) = @_;
  $self->{application} = $app;
  $self->setLabel( "Quit" );
  $self->whenPressed( sub { $self->{application}->exit(); } );
  return $self;
}
```

What's really great about these interpreted languages is that the code blocks come with a sort of "context." In the Smalltalk and Perl examples, the code block is eventually executed elsewhere, but runs as if it were still inside the object method. So, the code blocks can access all the data in the object, and any local variables in the particular code where they were created. In Python, the notation `variable.method(...)` executes a method, but `variable.method` creates a copy of the method that is bound to the object stored in `variable`. When it's executed later, it magically uses the object it's bound to for `self`.[2] In other languages, we have to use some other, clumsier method to ensure that the code block can access the information it needs.

In C, the only way to pass code around is with a function pointer, often called a "callback," which might look something like this (assuming we're using some object oriented GUI library such as the X toolkit):

[2] All this stuff about closures and scoping can be really mind-boggling when you just read it. Something that helps is to look at the examples above and figure out what has to happen for them to work.

```
void QuitButtonInitialize( Object * self, Application * app )
{
    Set( self, WIN_WHEN_PRESSED_CALLBACK, qbquit );
    Set( self, MY_APPLICATION, app );
}

void qbquit( Object * self )
{
    Application * app;

    app = (Application*) Get( self, MY_APPLICATION );
    AppClose( app );
}
```

Any information the "callback" function needs, such as the application, must be stored in the object itself, which in C tends to be a tangle of function calls and casting. There is no type checking going on (everything has to be in terms of `void *`'s). Despite the syntactical nightmare, exactly the same thing is accomplished as in the other examples.

What gets to be a problem is that not all C toolkits are object oriented. For example, the C library for Windows requires the programmer to assign a number to each event, then bind a callback to the number. When the callback is executed, it's passed a generalized pointer that must be decoded, which is bug prone and hard to do.

An improvement is a C++ wrapper framework around the underlying C library, such as Borland's Object Windows Library (OWL). OWL still uses the numbering mechanism, but it's almost invisible and most of the custom functionality is defined by subclassing and overriding a virtual function, rather than with a function pointer. This particular approach is a sort of Template Method, since the hard work is factored out in the superclass. Here's a pseudo-OWL subclass that works this way:

‡*Template Method, 325*

```
class QuitButton : public Button
{
private:
    Application & myApp;

protected:
    DECLARE_CALLBACK_TABLE; // a macro

public:
    QuitButton( Application & app ) : myApp( app )
    { SetLabel( "Quit" ); }

    // Override the ''when pressed'' function
    virtual void WhenPressed( void ) { myApp.Exit(); }
};

BEGIN_CALLBACK_TABLE(QuitButton) // more macros
BIND( BUTTON_PRESS_EVENT, WhenPressed )
END_CALLBACK_TABLE;
```

Alternatively, the Qt library defines some additions to the C++ language called signals and slots. Source code must pass through the Qt “meta-object compiler” which translates the additional keywords into regular C++. When a signal function is executed, all the slot functions it has been connected to are called. In this library, we can attach a response directly to the function that creates it:

```
class QuitButton : public Button
{
private:
    Application & myApp;

public slots:
    void Quit( void ) { myApp.Exit(); }

    QuitButton( Application & app ) : myApp( app )
    {
        SetLabel( "Quit" );
        Connect( SIGNAL( void ButtonPressed(void) ),
                 SLOT( *this, void Quit(void) ) );
    }
};

main()
{
    Application app;
    QuitButton quitButton( app );
}
```

The slot does not have to be in the same class as the signal. We could in fact do something like this:

```
class Application
{
public slots:
    void Exit( void );
};

main()
{
    Application app;
    Button quitButton;
    quitButton.SetLabel( "Quit" );
    quitButton.Connect( SIGNAL( void ButtonPressed(void) ),
                        SLOT( app, void Exit(void) ) );
}
```

Java does not provide for any sort of code block or even function pointers, but just about the same thing can be accomplished with a "Callback" interface defining a single member function. (We replace a function call based on a pointer with a virtual method.) So, we can write a callback class, and pass it to the button, like so:

```
interface Callback
{
    public void execute();
}

class QuitMe implements Callback
{
    private Application myApp;

    public QuitMe( Application app )
    {
        myApp = app;
    }

    public void execute()
    {
        myApp.exit();
    }
}


class QuitButton extends Button
{
    public QuitButton( Application app )
    {
        myApp = app;
        setLabel( "Quit" );
        whenPressed( new QuitMe( myApp ) );
    }
}

main()
{
    Application app = new Application;
    QuitButton quitButton = new QuitButton( app );
}
```

Java 1.1 provides a simple but strange-looking feature called inner classes that makes callbacks easier to write. Inner classes are defined inside of regular class definitions, and their instances are magically attached to an instance of the class they are inside. Furthermore, they have access to the private and protected members of the outer class.

```
class QuitButton extends Button
{
    private Application myApp;

    class QuitMe implements Callback
    {
        public void execute()
        {
            myApp.exit(); // Calls exit() through the implicit
                          // reference to the outer object.
                          // Same as:
                          // QuitButton.this.myApp.exit();
        }
    }

    public QuitButton( Application app )
    {
        myApp = app;
        setLabel( "Quit" );
        whenPressed( new QuitMe );
    }
}

main()
{
    Application app = new Application;
    QuitButton quitButton = new QuitButton( app );
}
```

As an alternative, the application class can contain the `QuitMe` inner class and provide a Factory method for constructing one. Then, whoever creates the button will have to bind the callback to the event:

```
class Application
{
    class QuitMe implements Callback
    {
        void execute()
        {
            exit(); // called on outer class
        }
    }

    Callback createQuitCallback()
    {
        return new QuitMe;
    }
    ...
}

main()
{
    Application app = new Application;

    Button quitButton( "Quit" );
    quitButton.whenPressed( app.createQuitCallback() );
}
```

The actual AWT uses a number of different callback interfaces, called "listeners," but this example illustrates the general idea.

4.3.4 Implementation

‡*Command, 233*

The solution used in Bargello is similar to the first Java example and follows the Command design pattern. C++ doesn't have any sort of code block or closure, so we have no choice but to use a class. C++ has no inner class concept, but that's really just a convenience in Java and not vital to how the pattern works.

To begin with, there is the `Command` class:

```
class Command
{
public:
    virtual void Execute( void ) = 0;
};
```

Simple enough. This corresponds to the `Callback` interface in the Java example. Additionally, we have:

```
template<class Type>
class ArgCommand
{
public:
    virtual void Execute( Type & arg ) = 0;
};
```

which is just a command object whose execution requires an argument. The trick now is to use `Commands` and `ArgCommand<Type>`'s throughout the rest of Bargello. For example, the command line parsing framework makes extensive use of commands.

4.3.5 Consequences

This particular solution causes a proliferation of classes. That is, each individual action has to be in its own class. That many classes can cause serious namespace pollution. A way around that is to use nested classes (not the same thing as inner classes) to hide the names of command classes inside the larger class that uses them.

On the other hand, functionality encapsulated in a command is not tied to any particular other large object and can easily be revised, exchanged for another command at run time, or used multiple times. For example, a quit button and a quit menu item could easily use the same command object, or at least instances of the same class to do their work. GUI toolkits which require us to subclass graphical components to customize their actions require additional design complexity. In the example, the `QuitButton` would have to know about `Applications` to be a separate subclass. If instead we hide that additional knowledge in a `Command` subclass, the button and application are decoupled, and there is no need to write a subclass of `Button`. Furthermore, there is no need to repeat the quitting code in a button class and a menu class and a hot-key class. . . .

One of the most powerful uses of commands is to implement undoable operations. In this case, the abstract command interface might have `Do()` and `Undo()` members. When a command is "done," it registers itself with a command list. We can undo the commands which have been executed so far by traversing the list in reverse and calling `Undo()` on each object. Additionally, we can redo the undone commands by going back the other way. . . .

Command objects can be implemented as Singletons, which is especially useful if they are shared.

‡*Singleton, 127*

Simple Factories can be thought of as special purpose commands with a function parallel to the `Execute()` function which creates an object of some kind.

‡*Factory, 87*

4.4 Parsing the command line

4.4.1 Intent

Provide a flxeible and powerful tool for parsing command lines, but to aslo include a means of making simple parsomg easy to do.

4.4.2 Formulation of the problem

In C++, a program is passed a list of strings typed on the command line through the arguments to `main()`. They are used to give it simple instructions and modify its behavior. Parsing the command line is often difficult because it's most convenient for the user to be able to enter flags and options in a fairly free format: flags can come in any order, any number of file names can be present, and so. The general pattern considered here is as follows:

- A special character at the beginning of a string, usually "`-`" or "`/`", indicates that it's a flag or option.
- A flag is a single string whose presence tells the program to take a certain action. For example, `-v` often puts a program into some sort of "verbose mode."
- An option is a string which considers the following string to be an argument. For example, `-o` ***filename*** usually means for the program to send its output to the given file instead of the default.
- Other strings are called "trailing strings" since they usually come at the end of the command line (but not always). They are often processed as a list of some kind.
- The special option `--` means the next string is a trailing string. This is important in case you want your program to deal with a file which happens to being with `-`.[3]

We can write a general parser which simply iterates over the strings and tests each to see if it's a flag, option, or trailing string, then selects some action from a table, and executes it. To be useful, we'll have to customize such a general-purpose parser in similar ways all the time. It makes sense to go ahead and provide that functionality in the library.

We will often want to interpret some command line strings as numerical values, as in `-depth 5`. So, the ability to process options with arguments of type `int` or `double` would make the parser more useful. Often, we simply want to assign a value to a variable, so we may as well factor out that code and put it in the library, too.

4.4.3 Cathedral pattern

One problem with library design is how to deal with excessive complexity. Often, a general purpose framework makes it easy to do difficult tasks: The hard work has already been done, and all we have to do is slide some custom objects into the gaps. However, many suck packages are complex and we have to read a lot of documentation to figure out they work. In the end, simple tasks are often as difficult to do as more complex ones. For example, in the Java 1.0 AWT, it was

[3]On UNIX, try removing a file named `-o` and you'll see why this is needed.

very easy to fetch a large picture file from a slow network connection while the rest of the program continued to work. When it was finished, it would sound an alarm, so to speak, and the program could use the image. However, simpler things like creating an image based on binary data, or reading one from a local file without using the alarm mechanism, were surprizingly difficult.

One way around this is to provide cathedral shaped frameworks. Imagine a gothic cathedral: The majority of the structure consists of large stones and buttresses, but there are lots of tiny details and decorations on top of them. We can apply the same principle of putting small components on top of large components in library design. At the heart is some very general, but possibly hard to use means of solving a problem. Implemented on top of that are successive layers of less general but more immediately useful functions. The framework then becomes a collection of small, manageable pieces, with larger, more flexible features available if they're needed.

4.4.4 Implementation of the command line parser

‡*Command, 233*

Custom operations are handled by the Command pattern, desribed in section 4.3. Commands are represented by objects that have an `Execute()` member function which performs an action. Simple commands which take no arguments implement the interface defined by class `Command`. Those which take a single argument are subclasses of `ArgCommand<Type>`.

Class `CommandLine` encapsulates the parsing code. It contains a map from flag names to their associated `Commands`, and a map from option names to their associated `ArgCommand<string>`'s.

The classes and their relationships are illustrated in figure 4.2.

Most of the real work is done by these two functions[4] which constititue the lowest layer:

```
class CommandLine
{
...
public:

    CommandLine & Flag( const string & name,
        const string & helpMessage,
                        MagicPointer<Command> command );
    CommandLine & Option( const string & name,
        const string & helpMessage,
        MagicPointer< ArgCommand<string> > command );
...
};
```

[4]Many of the functions in class `CommandLine` return `*this` so they may be chain called, as in `parser.Flag(...).Flag(...)...`.

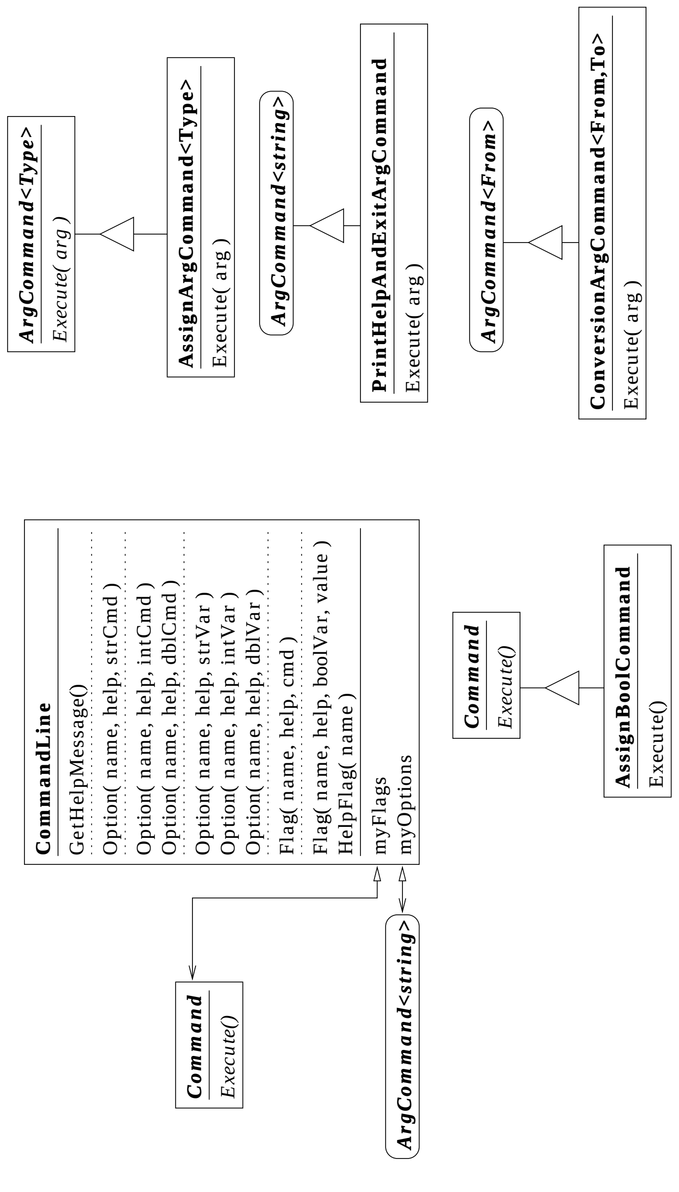

Figure 4.2: The command line parser and some of its helper classes. The dotted lines in class `CommandLine` separate the member functions into layers, as described in the text.

These bind a command object to a flag or option name. The parser also contains a help message which accumulates all the little messages for each flag or option. Notice that since the data structure is implemented in terms of classes `Command` and `ArgCommand<string>`, it is not possible to directly process a numerical argument to an option at this stage. Instead, we must write a special `ArgCommand<string>` class whose execute function converts its string argument to an `int` or `double`, then does something numerical with it.

Since converting an option's argument to a number is such a common task, it makes sense for it to be simple in the framework. So, we provide special `ConversionArgCommand<From,To>`'s which handle the conversion, then call an `ArgCommand<To>`. Note that these inherit from `ArgCommand<From>`, so we can instantiate conversion commands from `strings` to `ints` or `doubles`, and insert them into the table with the first-layer `Option` function described above. Rather than require programmers using the library to do all that construction themselves, we add the following additional versions of `Option()` which do it automatically:

```
class CommandLine
{
...
public:

    CommandLine & Option( const string & name,
        const string & helpMessage,
        MagicPointer< ArgCommand<int> > command );

    CommandLine & Option( const string & name,
        const string & helpMessage,
        MagicPointer< ArgCommand<double> > command );
...
}
```

Many times, all we want to do is assign the argument of an option to a variable. That's quite doable with something like `AssignArgCommand<Type>`, which is constructed with a reference to a variable and whose `Execute()` function assigns a new value to that variable. Again, rather than make applications programmers write their own `ArgCommands`, we can put them in the library and add three more versions of `Option()`. Class `AssignBoolCommand` fills in a similar role for boolean variables which are assigned based on the presence of flags. The interface to the parser now includes these methods:

```
class CommandLine
{
...
public:
    CommandLine & Flag( const string & name,
        const string & helpMessage,
        bool & var,
        bool newValue );

    CommandLine & Option( const string & name,
        const string & helpMessage,
        int & var );

    CommandLine & Option( const string & name,
        const string & helpMessage,
        double & var );

    CommandLine & Option( const string & name,
        const string & helpMessage,
        string & var );
...
}
```

The third layer consists of just one additional convenience function. All the functions so far include a help-string parameter. To print that out, we could create a `PrintHelpAndExitCommand` and attach it to a flag like `-h`, but since every program should at least be able to print out a command line help message, we may as well automate this, too:

```
class CommandLine
{
...
public:
    CommandLine & HelpFlag( const string & name );
...
};
```

Again, this function simply builds on the underlying abilities of class `CommandLine`.

4.5 Managing limited resources

4.5.1 Intent

To make a limited resource appear unlimited and save a lot of headaches.

4.5.2 Motivation

There is a small but difficult to solve problem that shows up in the most aggravating situations and adds complexity to what should be simple tasks: A program in UNIX is only allowed to have a fixed number of file handles open at any one time. File handles are not just for files. Directory traversal, standard input and output, and many other i/o operations require file handles.

A similar problem exists in many graphical user interface libraries: There can be no more than a handful of fonts available at once, for example.

Sometimes the same sort of problem appears in what would seem an entirely different situation: Only one operation can be performed on a hard drive or modem at once. These details become important when writing an operating system.

The problem here is that some sort of cricial resource is only available in limitied quantities and our programs must be able to operate when it runs out.

4.5.3 Generalization of the problem

All instances of a resource can be either open (in use) or closed (not in use). The limit is on the number of resources in use. The general problem here is to encapsulate the resource in such a way as to make it appear unlimited.

It's possible to connect all instances of a particular resource (file handles, GUI devices, etc.) so that when a new one is needed, but the supply has run out, another open instance can be temporarily closed to make room to open the new one.

4.5.4 Implementation

See figure 4.3 for the class structure of the resource framework. Every instance of a resource class must refer to a `ResourceRegistry`, which in turn refers to several objects which are open. When a resource object is opened, it informs its registry via the `Add()` function before attempting to acquire the resource. The registry keeps up with which resources are open in order of frequency of use. If needed, it closes the least frequently used resource before returning from the `Add()` method.

Whenever the resource object is told to perform some operation, such as the iteration methods in class `DirectoryStream`, the object first calls `Open()` to ensure that it's opened, then performs whatever operation it needs.

4.5.5 Writing concrete resource subclasses

A number of things are implemented by class `ResourceUser` which make subclasses easier to write. These are instances of the Template Method pattern, that is, member functions wihc do their work in terms of abstract functions which must be supplied by a concrete subclass.

‡ *Template Method, 325*

- The `ResourceUser` class, when constructed in the initializer list of a subclass, must be given a registry. A reference to it is stored automatically. It's often a singleton which is a private, static data member of the subclass. For example:

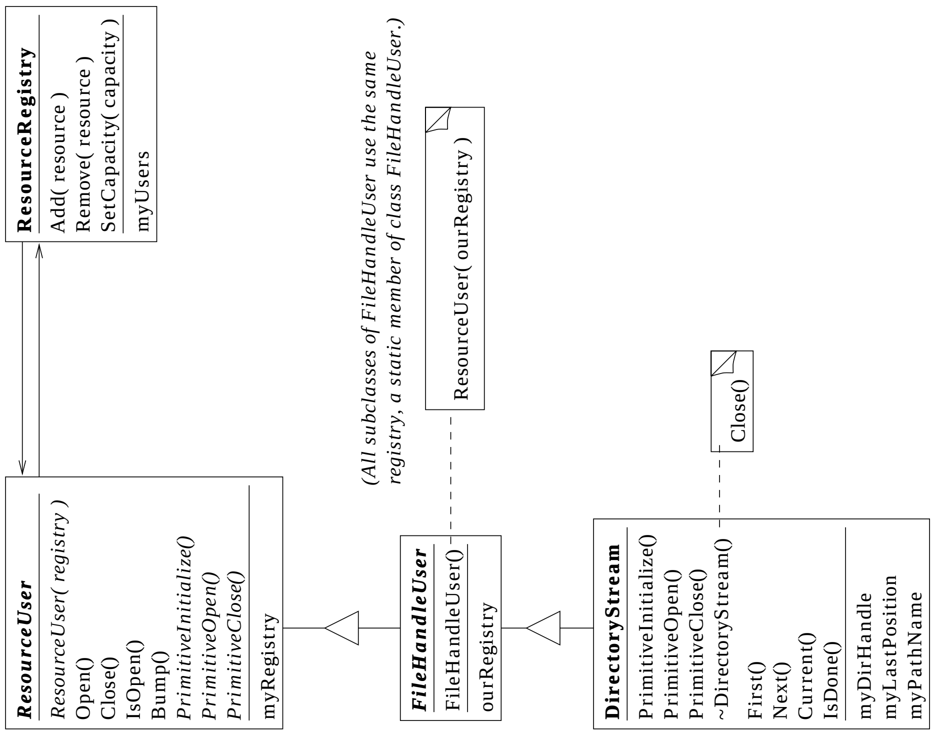

Figure 4.3: The Bargello resource framework

```
FileHandleUser::FileHandleUser( void )
    : ResourceUser( ourRegistry )
{
...
}
```

- `ResourceUser` implements `Open()` and `Close()` in terms of three abstract functions:
  - `PrimitiveInitialize()` is called only the first time the resource is opened. Unless you override it, it simply calls `PrimitiveOpen()`.
  - `PrimitiveOpen()` should acquire the resource and return it to its previously saved state. It will only be called if the object is currently closed.
  - `PrimitiveClose()` should save the state of the resource so it may be re-opened again later, then release the resource. It will only be called if the object is currently open.

  The `Open()` and `Close()` functions automatically deal with the registry. Calling `Open()` on an already-opened object is safe and does nothing. Likewise, calling `Close()` on an already-closed object is safe and does nothing.

There is one tricky thing to remember: The destructor in the concrete subclass absolutely must call `Close()`. It makes more sense to call `Close()` in the base class destructor; however, there is an obscure technicality in the C++ language which makes this impossible.[5]

4.5.6 Example

As a specific example, consider a `DirectoryStream`. It has a path name and a `DIR *` which points to a black-box directory stream data structure containing a file handle. Its state consists of the position in the stream where the next file name is to be read. The `PrimitiveInitialize()` implementation calls the system function `opendir()`. The `PrimitiveOpen()` implementation calls `opendir()`, then moves the stream forward to where it left off when it was last closed. `PrimitiveClose()` saves the `DIR *`'s current location, then calls `closedir()` to release the resource.

Since a `DIR *` internally uses a file handle, `DirectoryStream` inherits from `FileHandleUser` and registers all of its instances with a static registry in class `FileHandleUser`.

[5]If you're interested, this is what goes wrong. Take a `DirectoryStream` object, for example, and consider what happens when it goes out of scope. First, the `DirectoryStream` destructor is called, then the `FileHandleUser` destructor, and finally the `ResourceUser` destructor. If the `ResourceUser` destructor calls `Close()`, it eventually calls `PrimitiveClose()`, a virtual function. The program cannot now do the expected thing and use the definition in class `DirectoryStream`, because the `DirectoryStream` part of the object has already been destroyed. So, the program crashes.

4.6 The problem of persistent objects

4.6.1 Intent

A persistent object is one which can be written to and read from a data stream, such as a file or network socket. Persistent objects are a good way for programs to save documents and computed data. The Bargello library includes a framework which takes some of the dirty work out of persistence.

4.6.2 Motivation

Writing data is usually not difficult: a few calls to the i/o stream operators and the job is done. Reading it back in is the hard part. Unless the saved output is carefully formatted, reading will require a lot of complicated parsing, which often means code with lots of `if-else` chains and hard-to-follow flow control.

Reading data from a file when all the objects are basically the same is not too difficult. For example, reading in a matrix of real numbers in text format isn't hard at all. Nor is reading in a list of strings.

What gets tricky is reading and writing objects of different types to the same stream. For example, a drawing program must save text and polygons differently. Obviously, when reading a saved file back into memory, it must have some way of knowing the type of the next object in a data stream, preferably without a lot of parsing.

4.6.3 Solutions from other languages

Just for comparison, here are some ways the persistence problem has been solved in other languages and libraries.

In Smalltalk, an interpreted language, classes can define a `storeString` method which returns a string representation of the object. In particular, it must be a bit of Smalltalk code which can be executed to create an object with the same state as the original. Reading a data stream is then almost the same as running the interpreter. In C++ this would require creating a custom mini-language and accompanying interpreter, which can be a lot of work. There are some programs which save documents as LISP-like instructions or some other very simple language.

In Java, there is a serialization framework. The ability to save and restore primitive types is built into the language, and it's possible to store an object by storing the name and version of its class, and then storing all of its data members. Restoring it is then almost trivial. All of that functionality is part of the library. Since C++ classes are not so heavily automated, this method will not work without some adaptation.

4.6.4 Implementation

The solution used in Bargello is similar to the Java framework, but with differences due to the language and some efforts made to keep the size of the output from being too large. See figure 4.4.

Each concrete persistent class must implement a `StoreString()` method which returns a string representation of the object. Restoring the object from a stream is done using the Factory design pattern. Each persistent class must also provide a companion factory class which can create an object from its string representation. The factory is given a unique ID code when it is created. The factories are usually Singletons and *must be created in the same order* every time the program runs so they always get the same ID.[6]

The `PersistentArchive` class coordinates factories, objects, and data streams. To use a `PersistentArchive`, the program must do several things in a specific order:

- Create all the necessary factories in a fixed order.
- Create the archive itself. It must be given a name, which is used to verify objects when they are read back in.
- Register the appropriate factories with the archive. When the archive begins writing, it will list its own name, then all the factories which have been registered with it. Only objects associated with those factories can be written to the archive. Again, this is for verifying the data stream's correctness.
- Write persistent objects to the archive. They are written to the data stream in the form ***`idcode`***`:`***`length`***`:`***`representation`***`:`. So for example, the integer 761 might appear `107:3:761:`.
- Call `FinishWriting()` on the archive. This writes a sentinel value at the end.

Reading from a stream requires similar steps. The factories and then the `PersistentArchive` object must be created as before, and the factories registered with the archive. When the archive begins reading, it first verifies that the name at the head of the input stream matches its own, and that all the factories listed afterwards match up with the factories it knows about. Reading objects is just a matter of using the iterator methods of the archive and some type casting.

Internally, the archive reads each object as follows. First, the ID code is read in. Then, a factory registered with that ID code is selected. There is now a counted, delimited string at the front of the input stream. That string is read in and the selected factory is used to recreate the persistent object. A sentinel value indicates the end of the file.

4.6.5 One last unsolved problem

There's one last bit of functionality which is available in Java but not in the Bargello system. If two objects must both reference a third, the `Persistent`

[6]Unique ID codes are generated with a simple class that keeps a counter. Each time a new ID is needed, it bumps the counter and returns the next number.

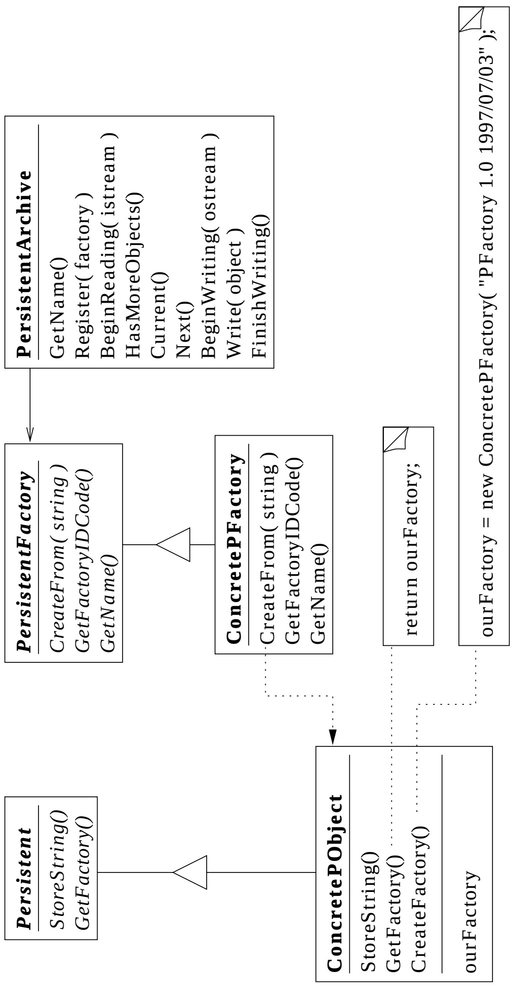

Figure 4.4: The Bargello persistency framework

framework is of no use for ensuring that the references are intact after the three are read from a stream. A persistent hash table of some kind might be useful for solving this problem.

4.7 The file system abstraction

4.7.1 Intent

The ability to scan through the contents of disk drives and other tree-shaped collections of files is generally useful and ought to be encapsulated in a flexible and powerful manner. The implementation used in the Bargello library achieves this through the use of the Visitor pattern and a number of creational patterns.

4.7.2 Motivation

There are three different kinds of files common to the platforms used by the Bargello library: plain files, which simply contain data; directories, also called folders, which contain a set of files; and links, also known as aliases and short-cuts, which refer to another file. Files which don't fit into one of these categories will be called "other files." All files have a name, a size in bytes, permissions (readable, writable, or executable), the time of the last change made to them, and a "real path" which is an absolute path to the file which doesn't contain any links.

A file system is a tree-shaped collection of files with a root directory and a current working directory. File systems can fetch information about a file from a path name. Relative path names are resolved based on the current directory. Note that these definitions are not restricted to disk drives. FTP sites and archive files may be treated like file systems as well.

Consider the following problems:

- List the contents of a directory in alphabetical order, omitting the `.` and `..` entries (which mean the current directory and its parent on UNIX and Windows).
- Recursively list the contents of a directory and all subdirectories within it, using depth first search, breadth first search, and in-order traversal.
- Create a snapshot of the state of a disk drive and use it later to locate all files which have changed, been added, or deleted since the snapshot was taken.
- Do the same logging and diffing for an anonymous FTP site.
- Do the same for a compressed archive (ie. a `tar.gz` file, a `ZIP` file, or a `Stuffit` file.)
- Locate all files in a certain directory which are larger than one megabyte and whose names contain the string "letter."

All of these tasks are essentially different versions of the same general problem of "file visitation," executing actions on some of the files in some sort of file system in a particular order. Since operating systems tend to provide a lot of powerful functions for dealing with file systems, it is tempting just to use a bunch of system calls to perform file visitations. However, this has the disadvantage of being very non-portable, and different operating systems make different parts of the task easy. For example, Windows automates the task of looking at just the files whose names match a particular pattern, such as `*.txt`, but UNIX does not. One solution is simply to wrap up the operating system calls in some sort of general-purpose Facade class which can be re-implemented on each platform in a different way.

There are two problems with that. The first is that the resulting class would be a "god class." For it to be capable of solving the above problems, it would have to support filtering, sorting, saving to some kind of data stream, restoring from a data stream, comparison to another such object, and three different forms of recursion. That's a lot of functionality for just one class. On top of that, the entire class must be re-implemented to deal with different kinds of file systems, such as FTP sites and archive files, for which the operating system doesn't provide helper functions. In short, this solution doesn't save a whole lot of work and is not reusable.

The second problem is that the class ends up being a mush of loosely related functions based on ideas borrowed from different libraries: Windows style filters, UNIX `scandir()` functionality, and so on. What should such a class be named? What abstraction does it represent? Is it difficult to document, and therefore difficult to understand and reuse?

The Bargello solution trades one kind of complexity (a single god class) for a different kind (a lot of little classes each with certain specific capabilities). The end result is more flexible and easier to maintain than the god class model.

4.7.3 Reformulation of the problem

Since file systems are structured like trees, it makes sense to represent them by a tree-like data structure. See the class diagram in figure 4.5 and a sketch of an object hierarchy in figure 4.6. Files are represented by different kinds of nodes, all of which inherit from class `FileNode`. Information about files usually comes from the operating system or some other source as a dumb data structure, so state information about files is encapsulated by a `FileInfo` object, one of which is stored in each file node.[7]

Each `DirectoryNode` has a child node for each file it contains. They are available through iteration methods. Note that `DirectoryNode::Current()` must return `FileNode&` because the directory can contain any sort of files an arbitrary order. The other node classes `PlainFileNode`, `LinkFileNode`, and

[7]This class could be eliminated by combining it with `FileNode`. However, it turns out to be useful to have all the state information of a file separate from the node functionality. `FileInfo` serves as a sort of Memento class which simplifies the `ShadowFileSystem` and construction of the `DiskFileSystem`, described elsewhere.

`OtherFileNode` represent the non-container files and make up the leaf nodes of the tree, that is, those which have no children.

The generalized problem of file visiting may now be stated more specifically: The library is responsible for providing abstract and concrete file system packages, including a means by which application programmers can apply custom operations to the contents of a file system. These custom operations must be able to include operating on a select subset of the contents of a directory in customizable order, so any extra functionality which facilitates these sorting and filtering operations should be included. also, anything which saves time would be advantageous, since most physical file systems are relatively slow.

4.7.4 Visitors

The Visitor pattern is an example of separating the part that changes from the part that doesn't change. In the generalized file visiting problem, the structure of a file system is not going to change. It will always be a tree with a few types of files. The order in which the files are traversed will change, however, as will which ones are visited, and what operations are performed on them. The Visitor pattern consists of a data structure containing different kinds of objects and a visitor class which encapsulates the operations, and in this case, visitation order.

Double dispatch

We want to be able to write code which will work on any file system, so there must be an abstract framework, and reusable code must be written in terms of it. But to maintain abstraction, the file system must deal with many different kinds of file nodes, so it must implement just about everything in terms of the base class `FileNode` and rely on virtual functions to deal with the specifics. The key to making visitors work is to be able to apply a function to any file node based on which specific class the file node belongs to (`Plain`, `Link`, `Directory`, or `Other`) and on the class a visitor object belongs to. The trick is called *double dispatch.*

Single dispatch means calling one out of many similar functions based on the type of one object. Not to be confused with overloaded functions, single dispatch is the idea behind virtual functions. If we declare `Gadget * gp` it could point to any object which belongs to class `Gadget` or any of its subclasses. If class `Gadget` includes a function `virtual void Open()`, then the statement `gp->Open()` calls whichever version of the `Open` function matches the type of the object pointed to by `gp`, not the declared type of `gp` itself. The same selection mechanism works on references, too.

Double dispatch is a generalization of the same thing: calling one out of many similar functions based on the types of two objects. It's not directly supported by C++ or most object oriented languages, but it can be done with some well-planned object interactions. This is how the Visitor pattern works.

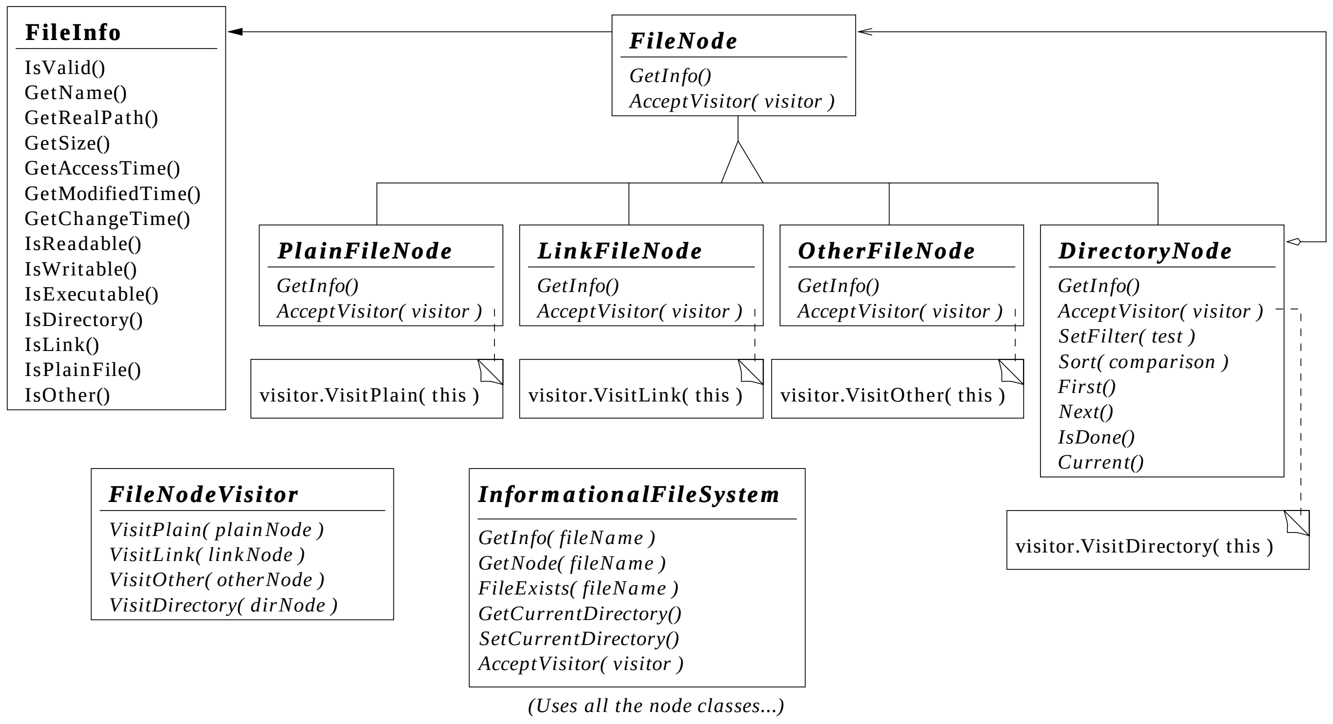

Figure 4.5: An abstract file system

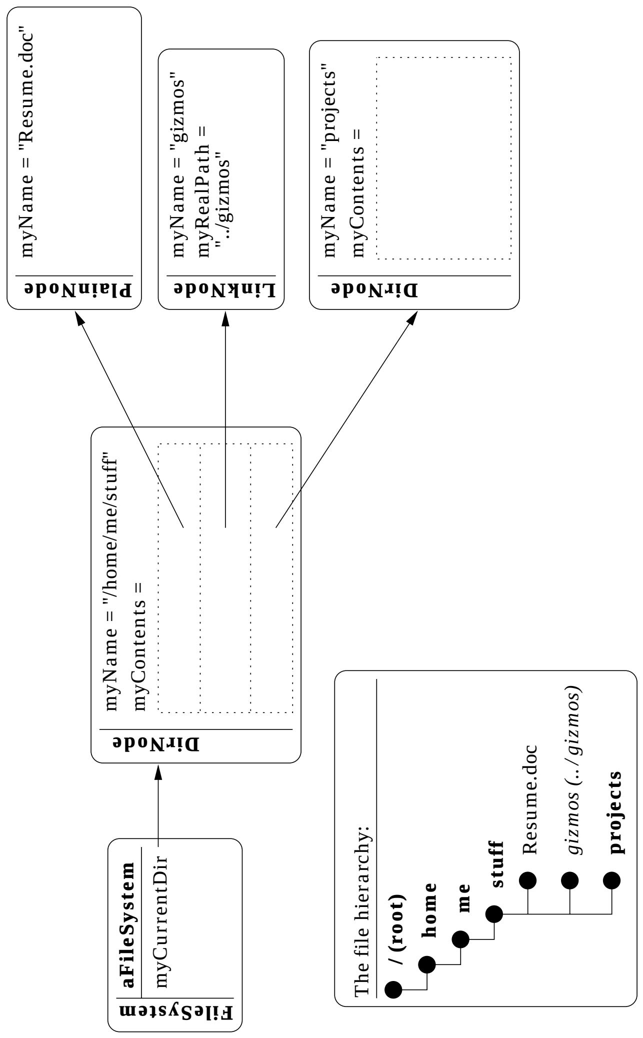

Figure 4.6: A sketch of the data structure.

Implementation

A different member function is declared in class `FileNodeVisitor` for every different sort of file:

```
class FileNodeVisitor
{
    ...
public:
    virtual void VisitPlain( PlainFileNode & ) = 0;
    virtual void VisitLink( LinkFileNode & ) = 0;
    virtual void VisitOther( OtherFileNode & ) = 0;
    virtual void VisitDirectory( DirectoryNode & ) = 0;
    ...
};
```

A concrete subclass must implement them. Since there can be any number of concrete subclasses, there can be any number of different versions of those operations.

All the different kinds of `FileNodes` implement a special `AcceptVisitor` function which takes a visitor for an argument and simple passes itself as an argument to one of its type specific functions. (This enables the visitor to "know" which type of node it's dealing with without a cast.):

```
PlainFileNode::AcceptVisitor( FileNodeVisitor & v )
{
    v.VisitPlain( *this );
}

LinkFileNode::AcceptVisitor( FileNodeVisitor & v )
{
    v.VisitLink( *this );
}

OtherFileNode::AcceptVisitor( FileNodeVisitor & v )
{
    v.VisitOther( *this );
}

DirectoryNode::AcceptVisitor( FileNodeVisitor & v )
{
    v.VisitDirectory( *this );
}
```

Now suppose we have `FileNode & fn` and `FileNodeVisitor & v` and we want to perform a different operation depending on what sort of visitor `v` refers to and where `fn` refers to a plain file, directory, or link (use double dispatch).

Since `FileNode` and `FileNodeVisitor` cooperate so well, this is done by calling `fn.AcceptVisitor( v )`. The particular `AcceptVisitor` function comes from whatever class `fn` refers to, and it in turn calls one of the type-specific `Visit...()` operations on `v`. The particular version of `Visit...()` is selected virtually, depending on the class `v` refers to.

Recursive traversal

In particular, to implement recursive traversal, the programmer must simply write a concrete subclass of `FileNodeVisitor` whose `VisitDirectory` function iterates over the files in the directory and calls `AcceptVisitor` on some of them, passing itself as the argument.

4.7.5 A few details about object creation

Nodes for plain files and links are created by the concrete file system class itself. Nodes for directories are a little more complicated.

The `DirectoryNode` class acts like a container for file nodes, since it must provide iteration, sorting, and filtering functionality. The problem is, where does it get its contents?

One solution is that it should fill itself up when it is created. The problem with that is that most file systems, such as disk drives and FTP sites, are relatively slow. If each directory node were filled upon construction, then the entire file hierarchy would have to be read in whenever a file system was created. That can take considerable time.

The solution used in Bargello is to provide for lazy initialization. `DirectoryNode` has functions not listed in the diagram for adding file nodes to it, and specifying that it has been fully constructed. However, if another object starts to iterate over a `DirectoryNode` and it has not been filled yet, the `DirectoryNode` calls a set of "primitive iteration" functions to fill itself before continuing. That is, it can wait to collect the information until someone needs it. Concrete subclasses of `DirectoryNode` must either override the primitive iteration functions, or be constructed in such a way that they are never called.

The primitive iteration functions are a modified form of the Factory Method design pattern.

4.7.6 The `DiskFileSystem`

Since the classes described so far are all abstract, it is necessary to define concrete subclasses for a particular type of file system. The case of a physical disk file system is the most obvious. See figure 4.7 for the `DiskFileSystem` framework.

Class `DiskFileSystem` serves as a file node Factory. It makes system calls (through auxiliary classes) which fetch information about files on disk and uses that data to create `DiskPlainFileNodes` and so on. `DiskDirectoryNodes` also know which file system they belong to and use it to create their child nodes on

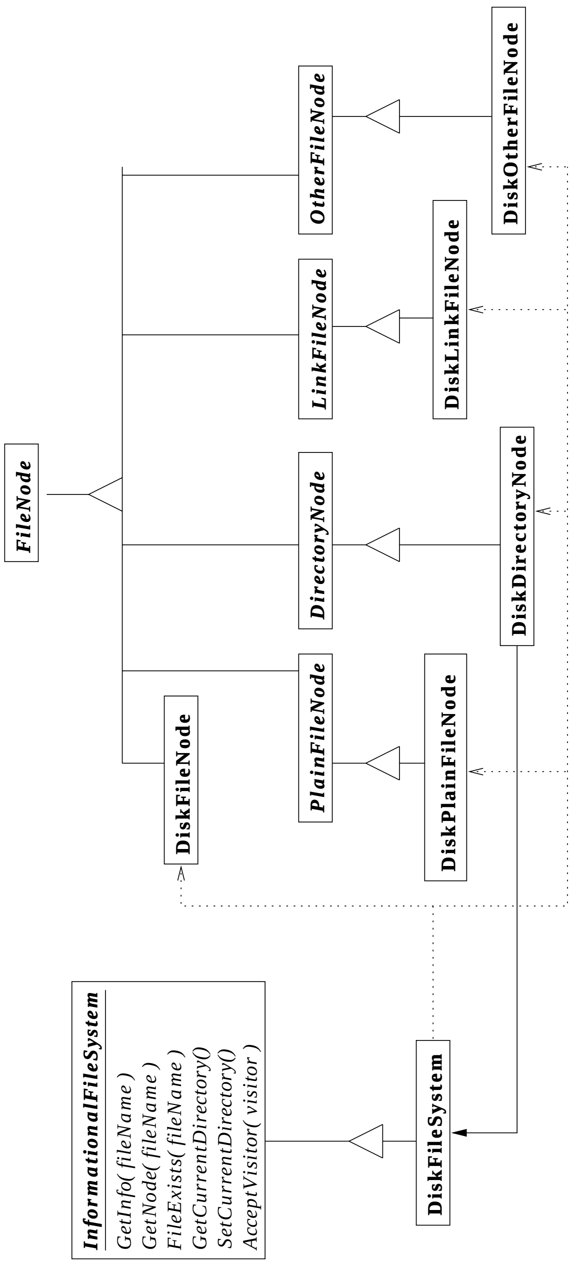

Figure 4.7: A physical file system

demand, thereby taking advantage of the lazy initialization mechanism provided in the superclass.

4.7.7 The `ShadowFileSystem`

In the original formulation of the problems for this section, several of the tasks included saving information to a data stream for use later on. So, how can that be done?

One solution is to use make all file systems persistent (capable of being written to and read from data streams). The problem with that is that the `DiskFileSystem` would have to provide two types of functionality: It would serve as a Facade and Adaptor by encapsulating operating system calls, and also as a container for the retrieved information. (It has to contain the data. It can't write it back to the physical device it came from, and usually that's not what you want it to do anyway.) The same difficulty would plague an FTP file system, or a compressed archive file system.

A better idea is to have another kind of `InformationalFileSystem` which is persistent and serves only to store information about another file system. In Bargello, this functionality is provided by class `ShadowFileSystem` and its associates.

The class diagram for `ShadowFileSystem` is very much like the one for `DiskFileSystem`, so it isn't repeated. The main differences are that the four kinds of shadow file nodes and the file system itself are all persistent, and in how `ShadowFileSystems` are created.

There is no way around directly storing all the data required by the file system, unlike the `DiskFileSystem` which could use lazy initialization and delete the contents of directories which had already been traversed. So, `ShadowFileSystem` has methods for adding information to itself and behaves more like a container class. When an instance is initially created, it must be given another `InformationalFileSystem` which it will "shadow," or copy data from. To get the data, a special creation visitor is applied to the other file system which traverses it recursively and adds a shadow version of each file node to the `ShadowFileSystem` under construction.

To save time, the entire file system is not shadowed. Only the directories needed to construct a path to the current directory, the current directory, and all files contained in it (recursively) are shadowed. For example, if you create a `DiskFileSystem dfs` and set its current directory to `/home/me`, a `ShadowFileSystem` constructed from it would contain `/`, `/home`, `/home/me`, and everything inside of `home/me`, but not `/usr` or `/home/otherguy`. If you need to shadow an entire file system, just be sure to change to its root directory before constructing the shadow.

`ShadowFileSystems` and the companion node classes all utilize Bargello's persistency framework to save themselves to and restore themselves from data streams.

4.8 The remaining Oodle classes

There remains now jus the tast of combining all those library classes into a program. There are three Oodle specific classes, shown in figure 4.8.

`NewListVisitors` are created with a file system which they refer to as a "master copy." When they visit a file node, they check to see if it exists in the master copy. If not, then it is considered new and printed out.

`ChangedListVisitor` does almost the same thing, but it outputs only those files which have a newer date than the corresponding file in the master copy.

‡*Mediator, 273*

The application class is `TextOodleApp`. It runs Oodle using a text-only command line interface. It coordinates the other classes, thereby acting as a sort of Mediator. Its `Log()` function creates a `DiskFileSystem` and a shadow of it, then writes the shadow to a file. Its `Diff()` function creates a disk file system and reads a shadow file system from a log file, then applies several visitors to generate the output. It creates a `NewListVisitor` whose master copy is the shadow and sends it to visit the disk file system, thereby printing out the files which have been created since the last log. It creates another `NewListVisitor` whose master copy is the disk file system and sends it to visit the shadow, thereby printing out all the files which have been deleted since the log. Note how nicely this reversal of viewpoint solves two parts of our problem with minimal duplicated code. The final step is to create a `ChangedListVisitor` and send it visiting the disk file system.

Note that none of these classes does much work. All of that is hidden away in the library somewhere. If someone got adventurous and write an FTP file system or a TAR file system, it would be very easy to adapt these three classes to take advantage of it. The interface is tightly woven into all three classes, but they are so trivial that is't pointless to separate out any more functionality until we have definite plans for a more powerful interface.

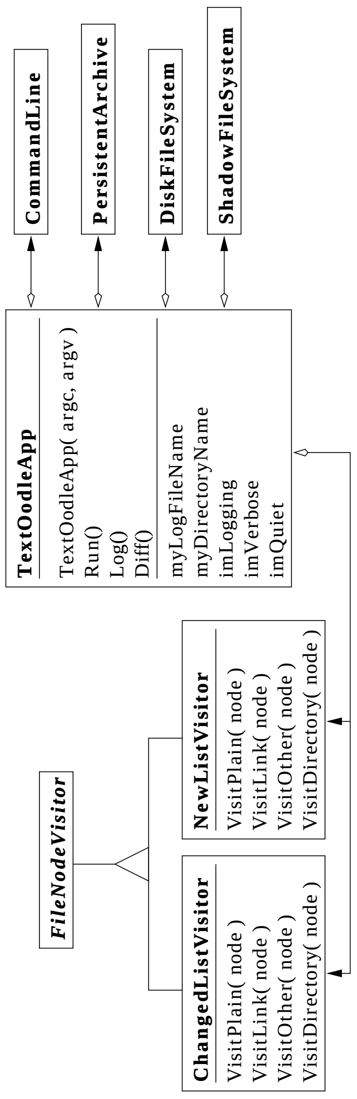

Figure 4.8: The Oodle application design.

Bibliography

[1] Owen Astrachan. *A Computer Science Tapestry*. McGraw-Hill, 1997.

[2] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.