,

# Python 3 Patterns, Recipes and Idioms

***Release 1.0***

**Bruce Eckel**

**May 10, 2017**

# Contents

CHAPTER 1

# Contributors

List of contributors.

**Note:** This needs some thought. I want to include everyone who makes a contribution, but I'd also like to indicate people who have made larger contributions – there's no automatic way to do this now that we have moved to BitBucket and are using Wikis to allow people to make contributions more simply in the beginning.

## Thanks To

- BitBucket.org and the creators of Mercurial
- Creator(s) of Sphinx
- And of course, Guido and the team for their incessant improvement of Python, especially for taking the risk in breaking backward compatibility in Python 3.0 to refactor the language.

**Todo**

Yarko (example label of ToDo):

- update CSS styles for todo's & todo lists;
- look at http://sphinx.pocoo.org/ext/coverage.html for example.
- Autogenerated ToDoLists do not appear in LaTeX output - debug, fix;
- DONE: - ToDo does not appear to be created by make dependencies (it's autogenerated); - update Makefile to always re-generate todo lists;

CHAPTER 2

# ToDo List

Currently, this doesn't seem to link into the index, as I'd hoped.

- Refine "Printed Book" and "Translations"
- Code extractor for rst files (maybe part of intro chapter?)
- Code updater to put code in/refresh code into book.
- Move frontmatter into its own directory
- <!> Seems to be a warning sign (but initial tests didn't work)
- Idea: decorator on a dictionary object, to turn it into an ordered dictionary.
- "Other resources" at the end of each chapter
- For print version, convert hyperlinks into footnotes.
  - build tool for this, or check int rst handling of this - see if it works with Sphinx;

# CHAPTER 3

The remainder are from context, from the book.

**Todo**

Yarko (example label of ToDo):

- update CSS styles for todo's & todo lists;
- look at http://sphinx.pocoo.org/ext/coverage.html for example.
- Autogenerated ToDoLists do not appear in LaTeX output - debug, fix;
- DONE: - ToDo does not appear to be created by make dependencies (it's autogenerated); - update Makefile to always re-generate todo lists;

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-3-patterns-idioms-test/checkouts/latest/docs/Contributors.rst, line 27.)

**Todo**

The remainder of this document needs rewriting. Rewrite this section for BitBucket & Mercurial; make some project specific diagrams;

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-3-patterns-idioms-test/checkouts/latest/docs/DeveloperGuide.rst, line 138.)

**Todo**

This section still work in progress:

- `hg branch lp:python3patterns`
- `hg commit -m 'initial checkout'`
- (hack, hack, hack....)
- `hg merge` (pull new updates)

- `hg commit -m 'checkin after merge...'`
- ... and so on...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-3-patterns-idioms-test/checkouts/latest/docs/DeveloperGuide.rst, line 207.)

**Todo**

Someone who knows more about emacs for Linux please add more specific information about the windowed version(s).

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-3-patterns-idioms-test/checkouts/latest/docs/DeveloperGuide.rst, line 302.)

**Todo**

rewrite to distinguish python generator from above description, or choose different name.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-3-patterns-idioms-test/checkouts/latest/docs/Jython.rst, line 493.)

CHAPTER 4

# A Note To Readers

What you see here is an early version of the book. We have yet to get everything working right and rewritten for Python. Or even to get the book testing systems in place.

If you're here because you're curious, that's great. But please don't expect too much from the book just yet. When we get it to a point where everything compiles and all the Java references, etc. have been rewritten out, then this note will disappear. Until then, *caveat emptor*.

CHAPTER 5

# Introduction

## A Team Effort

This book is an experiment to see if we can combine everyone's best efforts to create something great.

You can find the contributors right before this introduction. They are listed in order of *Karma Points*, a system Launchpad.net uses to keep track of contributions from everyone working on an open-source project.

In my case, I will write new material, as well as rewriting other contributions to clarify and give voice, setting up the architecture and being the *Benevolent Dictator* for the book. But I definitely won't be doing everything; my goal is that this is a team project and that everyone who wants to will have something useful to contribute.

We'll be using Launchpad.net's "Blueprints" facility to add "features" to the book, so that's where you'll find the main repository of things to do.

What can you contribute? Anything as small as spelling and grammatical correctons, and as large as a whole chapter. Research into new topics and creating examples is what takes me the most time, so if you know something already or are willing to figure it out, don't worry if your writing or programming isn't perfect – contribute it and I and the rest of the group will improve it.

You may also have talents in figuring things out. Sphinx formatting, for example, or how to produce camera-ready formatting. These are all very useful things which will not only benefit this book but also any future book built on this template (every bit of the build system for the book will be out in the open, so if you want to take what we've done here and start your own book, you can).

Of course, not everything can make it into the final print book, but things that don't fit into the main book can be moved into an "appendix book" or a "volume 2" book or something like that.

## Not an Introductory Book

Although there is an introduction for programmers, this book is not intended to be introductory. There are already lots of good introductory books out there.

You can think of it as an "intermediate" or "somewhat advanced" book, but the "somewhat" modifier is very important here. Because it is not introductory, two difficult constraints are removed.

1. In an introductory book you are forced to describe everything in lock step, never mentioning anything before it has been thoroughly introduced. That's still a good goal, but we don't have to agonize over it when it doesn't happen (just cross-reference the material).
2. In addition, the topics are not restricted; in this book topics are chosen based on whether they are interesting and/or useful, not on whether they are introductory or not.

That said, people will still be coming to a topic without knowing about it and it will need to be introduced, as much as possible, as if they have never seen it before.

## The License



If you make contributions, you must own the rights to your material and be able to place them under this license. Please don't contribute something unless you are sure this is the case (read your company's employment contract – these often specify that anything you think of or create at any time of day or night belongs to the company).

## The Printed Book

Because of the creative commons license, the electronic version of the book as well as all the sources are available, can be reproduced on other web sites, etc. (again, as long as you attribute it).

You can print your own version of the book. I will be creating a printed version of the book for sale, with a nice cover and binding. Many people do like to have a print version of the book, and part of the motivation for doing a print version is to make some income off the effort I put into the book.

But buying my particular print version of the book is optional. All of the tools will be downloadable so that you can print it yourself, or send it to a copy shop and have it bound, etc. The only thing you won't get is my cover and binding.

## Translations

Launchpad.net, where this project is hosted, has support for doing translations and this gave me an idea. I had just come back from speaking at the Python conference in Brazil, and was thinking about the user group there and how I might support them. (We had done a seminar while I was there in order to help pay for my trip and support the organization).

If the book can be kept in a Sphinx restructured text format that can be turned directly into camera-ready PDF (the basic form is there but it will take somebody messing about with it to get it into camera-ready layout), then the job of translation can be kept to something that could be done by user groups during sprints. The user group could then use a print-on-demand service to print the book, and get the group members to take them to local bookstores and do other kinds of promotions. The profits from the book could go to the user group (who knows, just like the Brazillian group, your group may end up using some of those profits to bring me to speak at your conference!).

If possible, I would like a royalty from these translations. To me, 5% of the cover price sounds reasonable. If the user group would like to use my cover, then they could pay this royalty. If they wanted to go their own way, it's creative commons so as long as the book is attributed that's their choice.

## My Motives

Just so it's clear, I have the following motives for creating this book:

1. Learn more about Python and contribute to the Python community, to help create more and better Python programmers.
2. Develop more Python consulting and training clients through the publicity generated by the book (see here).
3. Experiment with group creation of teaching materials for the book, which will benefit me in my own training (see the previous point) but will also benefit anyone choosing to use the book as a text in a course or training seminar. (See *Teaching Support*).
4. Generate profits by selling printed books. (But see above about the ability to print the book yourself).
5. Help raise money for non-U.S. Python user groups via translations, from which I might gain a small percentage.

CHAPTER 6

# Teaching Support

Teachers and lecturers often need support material to help them use a book for teaching. I have put some exercises in, and I hope we can add more.

I'd also like to create teaching materials like slides and exercises as a group effort with a creative commons license, especially by allying with those people who are in the teaching professions. I'd like to make it something that everyone would like to teach from – including myself.

Here are some places to get ideas for exercises and projects:

- For coding dojos
- Rosetta Code

CHAPTER 7

# Book Development Rules

Guidelines for the creation process.

**Note:** This is just a start. This document will evolve as more issues appear.

## Contribute What You Can

One of the things I've learned by holding open-spaces conferences is that everyone has something useful to contribute, although they often don't know it.

Maybe you're don't feel expert enough at Python to contribute anything yet. But you're in this field, so you've got some useful abilities.

- Maybe you're good at admin stuff, figuring out how things work and configuring things.
- If you have a flair for design and can figure out Sphinx templating, that will be useful.
- Perhaps you are good at Latex. We need to figure out how to format the PDF version of the book from the Sphinx sources, so that we can produce the print version of the book without going through hand work in a layout program.
- You probably have ideas about things you'd like to understand, that haven't been covered elsewhere.
- And sometimes people are particularly good at spotting typos.

## Don't Get Attached

Writing is rewriting. Your stuff will get rewritten, probably multiple times. This makes it better for the reader. It doesn't mean it was bad. Everything can be made better.

By the same measure, don't worry if your examples or prose aren't perfect. Don't let that keep you from contributing them early. They'll get rewritten anyway, so don't worry too much – do the best you can, but don't get blocked.

There's also an editor who will be rewriting for clarity and active voice. He doesn't necessarily understand the technology but he will still be able to improve the flow. If you discover he has introduced technical errors in the prose, then feel free to fix it but try to maintain any clarity that he has added.

If something is moved to Volume 2 or the electronic-only appendices, it's just a decision about the material, not about you.

## Credit

As much as possible, I want to give credit to contributions. Much of this will be taken care of automatically by the Launchpad.net "Karma" system. However, if you contribute something significant, for example the bulk of a new chapter, then you should put "contributed by" at the beginning of that chapter, and if you make significant improvements and changes to a chapter you should say "further contributions by" or "further changes by", accordingly.

## Mechanics

- Automate everything. Everything should be in the build script; nothing should be done by hand.
- All documents will be in Sphinx restructured text format. Here's the link to the Sphinx documentation.
- Everything goes through Launchpad.net and uses Launchpad's Bazzar distributed version control system.
- Follow PEP8 for style. That way we don't have to argue about it.
- Camelcasing for naming. PEP8 suggests underscores as a preference rather than a hard-and fast rule, and camelcasing *feels* more like OO to me, as if we are emphasizing the design here (which I want to do) and putting less focus on the C-ish nature that *can* be expressed in Python.

```
The above point is still being debated.
```

- Four space indents.
- We're not using chapter numbers because we'll be moving chapters around. If you need to cross-reference a chapter, use the chapter name and a link.
- Index as you go. Indexing will happen throughout the project. Although finalizing the index is a task in itself, it will be very helpful if everyone adds index entries anytime they occur to you. You can find example index entries by going to the index, clicking on one of the entries, then selecting "view source" in the left-side bar (Sphinx cleverly shows you the Sphinx source so you can use it as an example).
- Don't worry about chapter length. Some chapters may be very small, others may be quite significant. It's just the nature of this book. Trying to make the chapters the same length will end up fluffing some up which will not benefit the reader. Make the chapters however long they need to be, but no longer.

## Diagrams

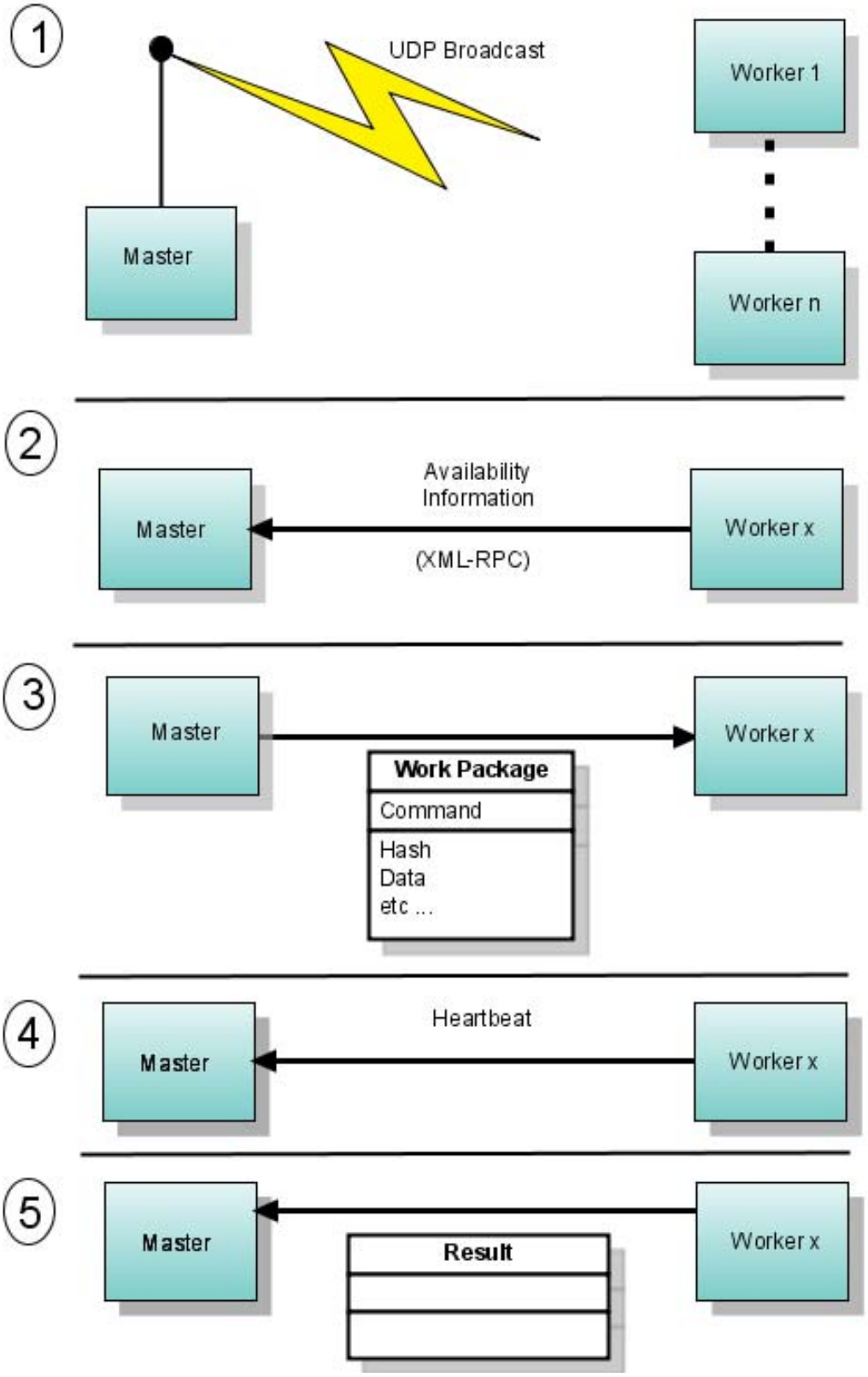
Create diagrams using whatever tool is convenient for you, as long as it produces formats that Sphinx can use.

It doesn't matter if your diagram is imperfect. Even if you just sketch something by hand and scan it in, it will help readers visualize what's going on.

At some point, diagrams will be redone for consistency using a single tool, with print publication in mind. This tool may be a commercial product. However, if you need to change the diagram you can replace it with your new version using your tool of choice. The important thing is to get the diagram right; at some point it will be redone to look good.

Note that all image tags should use a `*` at the end, not the file extension name. For example `..image:: _images/foo.*`. This way the tag will work for both the HTML output and the Latex output. Also, all images should be placed in the `_images` directory.

Here's an example which was done with the free online service Gliffy.com, then modified using the free Windows program Paint.NET (note, however, that we should not use color because it won't translate well to the print book):


1
UDP Broadcast
Worker 1
Master
Worker n
2
Availability
Information
Master
Worker x
(XML-RPC)
3
Master
Worker x
Work Package
Command
Hash
Data
etc ...
4
Heartbeat
Master
Worker x
5
Master
Worker x
Result

CHAPTER 8

# Developer Guide

Details for people participating in the book development process.

## Getting Started: The Easiest Approach

If all of the details are a little overwhelming at first, there's an easy way for you to make contributions without learning about distributed version control and Sphinx:

1. Create an account at http://www.BitBucket.org.
2. In your account, you get a wiki. In your wiki, create a page for your contribution. Just add your code and descriptions using plain text.
3. Point us to your wiki via the newsgroup.
4. We'll take your contribution and do the necessary formatting.

If you want to take another step, you can learn how to format your wiki page using Sphinx by looking at the source for pages in the book. You can try it right now – on the left side of this page (in the HTML book) you'll see a header that says **This Page** and underneath it **Show Source**. Click on **Show Source** and you'll see the Sphinx source for this page. Just look at the page sources and imitate that.

When you're ready, you can learn more about Sphinx and Mercurial and begin making contributions that way.

The following sections are for those who are ready to build the book on their own machines.

## For Windows Users

You need to install Cygwin; go to:

http://www.cygwin.com

You need to install at least the `make` utility, but I find that `chere` (command prompt here) is also very useful.

Also install `openssh` (under **Net**), so you can create your RSA key for Mercurial.

I've discovered that it's best if you *don't* install Python as part of Cygwin; instead use a single Python installation under windows. Cygwin will find the installation if it is on your Windows PATH.

Because of this, you shouldn't select "mercurial" when you're installing Cygwin because that will cause Python to be installed. Instead, install them as standalone Windows applications (see below).

## Installing Sphinx

Because we are sometimes pushing the boundaries of Sphinx, you'll need to get the very latest development version (a.k.a. the "tip").

1. Get mercurial:

   http://www.selenic.com/mercurial

   Avoid installing the tortoiseHG part - it has caused trouble w/ Python debuggers.

2. To get the Sphinx trunk, start with:

   ```
   $ hg clone http://www.bitbucket.org/birkenfeld/sphinx/
   ```

   and to update, use:

   ```
   $ hg pull
   ```

   Once you update, run

   ```
   $ python setup.py install
   ```

   (You can repeat this step whenever you need to update).

   We may talk about minimum version numbers to process the book. Check your version with:

   ```
   $ hg identify -n
   ```

The full anouncement from Georg (Sphinx creator) is here:

http://groups.google.com/group/sphinx-dev/browse_thread/thread/6dd415847e5cbf7c

Mercurial Cheat sheets & quick starts should be enough to answer your questions:

- http://edong.net/2008v1/docs/dongwoo-Hg-120dpi.png
- http://www.ivy.fr/mercurial/ref/v1.0/

## Getting the Development Branch of the Book

This book uses BitBucket.org tools, and additional tools if necessary.

1. Sign up for an account at http://BitBucket.org.
2. You must create an rsa key. Under OSX and Linux, and if you installed `openssh with` Cygwin under windows, you run `ssh-keygen` to generate the key, and then add it to your BitBucket account.
3. Go to http://www.bitbucket.org/BruceEckel/python-3-patterns-idioms/, and you'll see instructions for getting a branch for development.

4. Work on your branch and make local commits and commits to your BitBucket account.

## Building the Book

To ensure you have Cygwin installed correctly (if you're using windows) and to see what the options are, type:

```
make
```

at a shell prompt. Then you can use `make html` to build the HTML version of the book, or `make htmlhelp` to make the windows help version, etc.

You can also use the `build` system I've created (as a book example; it is part of the distribution). This will call `make` and it simplifies many of the tasks involved. Type:

```
build help
```

to see the options.

---

**Todo**

The remainder of this document needs rewriting. Rewrite this section for BitBucket & Mercurial; make some project specific diagrams;

---

## Building the PDF

In order to build the Acrobat PDF verion of the book, you must install some additional software:

**Mac OSX**: Install the http://www.tug.org/mactex/ distribution. Although this is a Mac installer, it installs all the necessary command-line binaries to create the PDF of the book, and modifies your PATH variable.

**Windows**: Install following these instructions: http://www.tug.org/texlive/windows.html

**Linux**: Your Linux install may already have support, but if not, install following these instructions: http://www.tug.org/texlive/quickinstall.html

Once TeX is installed, move to this book's **src** directory and run `make latex`. When that command runs successfully, it will give you instructions as to how to finish.

## Setting up Mercurial

It's easier if you put a configuration file called **.hgrc** in your home directory. Here's one that sets up the user name and configures **kdiff3** as the diff tool for Mercurial to use when showing you differences between files:

```
# This is a Mercurial configuration file.
[ui]
username = Firstname Lastname <email@mailer.net>

[merge-tools]
# Override stock tool location
kdiff3.executable = /usr/bin/kdiff3
# Specify command line
```

```
kdiff3.args = $base $local $other -o $output
# Give higher priority
kdiff3.priority = 1
```

In addition, you can change the editor that Mercurial uses via an environment variable. For example, on OSX and Linux (and Windows with cygwin) you add this to your **.bash_profile** to set **emacs** as the default editor:

```
export set EDITOR=/usr/bin/emacs
```

## Working with BitBucket and Mercurial

**Note:** Adapted from a posting by Yarko Tymciurak

This assumes that you have created a local branch on your private machine where you do work, and keep it merged with the trunk.

That is, you've done:

- Forked a branch of http://www.bitbucket.org/BruceEckel/python-3-patterns-idioms/ (the main trunk; this fork will provide a place for review and comment)
- cloned the trunk to your local machine: - hg clone https://my_login@bitbucket.org/BruceEckel/python-3-patterns-idioms/
- cloned your local copy of trunk to create a working directory: - hg clone python-3-patterns-idioms devel

**Todo**

This section still work in progress:

- `hg branch lp:python3patterns`
- `hg commit -m 'initial checkout'`
- (hack, hack, hack....)
- `hg merge` (pull new updates)
- `hg commit -m 'checkin after merge...'`
- ... and so on...

When you have a new function idea, or think you've found a bug, ask Bruce on the group.

- If you have a new feature, create a wiki page on BitBucket and describe what you're going to do.
- If you have found a bug, make a bug report on BitBucket (later assign it to yourself, and link your branch to it);
- If you want to work on a project, look for an unassigned bug and try to work it out - then proceed as below...

When you are ready to share your work have others review, register a branch.

**Note:** You can re-use one branch for multiple bug fixes.

1. Sign up for an account on BitBucket.org
2. Go to the project and select "register branch" (`https://code.BitBucket.org/python3patterns/+addbranch`). Suggest you create a hosted branch, then you can work locally, and pull/push as you make progress (see http://doc.Mercurial-vcs.org/latest/en/user-guide/index.html#organizing).
3. Once you have registered your branch, BitBucket will provide you with instructions on how to pull and push to your personal development copy.
4. Link your bug report or blueprint to your branch.
5. Merge from your "parent" (the trunk, or others you are working with) as needed.
6. Push your working copy to BitBucket as your work is ready for others to review or test.
7. Once you are done making your changes, have completed testing, and are ready for the project team to inspect & test, please select "propose for merging"
8. Somebody on the core team will make a test merge (it may include merging with other patches). Once tests pass, and your branch is accepted, it will be merged into the trunk.

## A Simple Overview Of Editing and Merging

1. `hg pull http://www.bitbucket.org/BruceEckel/python-3-patterns-idioms/`
2. `hg merge` This brought up kdiff3 (note: this requires a separate installation of **kdiff3**)on any file's w/ conflicts, and you get to just visually look - left-to-right at A:base, B:mine, and C:yours.... the NICE thing is when you want BOTH the other and yours, you can click BOTH B & C buttons — sweeet! you can also review the "automatic" merges, choose which - conflicts only, or any merge.
3. ... `make html; make latex` ..... look at outputs (simultaneously, comparatively)... make any changes.... repeat....
4. `hg ci` without a message, it brought up an editor with a list of all changed files - so you can comment individually.

## Emacs for Editing Restructured Text

If you want an editing system with support for restructured text, one choice is the free text editor *emacs*, which has an add-on mode for restructured text. Emacs has a long and venerable history, and is an extremely powerful editor. Emacs also has versions that are customized for operating systems to make it much more familiar.

Here's a simple introduction to emacs and a useful introductory help guide. For Windows, there's a special FAQ.

**Mac OSX**: Comes with built-in emacs which you can invoke from the command line. For a nicer version, install Aquamacs, which looks and feels like a native Mac application.

**Windows**: You can download the latest windows installer here (choose the highest numbered zip file with "bin" in the name). This blog gives useful tips to make emacs on Windows even friendlier (in particular, it puts emacs on the right-click menu and improves the startup settings).

**Linux**: It's virtually guaranteed that you already have emacs preinstalled on your Linux distribution, which you can start from a command prompt. However, there may also be more "windowy" versions that you can install separately.

---

**Todo**

Someone who knows more about emacs for Linux please add more specific information about the windowed version(s).

---

Finally, here's the documentation for installing and using the emacs restructured-text mode. The elisp code it refers to is in the file rst.el.

To customize your emacs, you need to open the `.emacs` file. The above Windows FAQ tells you how to put your `.emacs` file somewhere else, but the easiest thing to do is just open emacs and inside it type `C-x C-f ~/.emacs`, which will open your default `.emacs` file if you have one, or create a new one if you don't.

You'll need to install **rst.el** someplace emacs will find it. Here's an example **.emacs** file which adds a local directory called **~/emacs/** to the search path, (so you can put **.el** files there) and also automatically starts **rst** mode for files with extensions of **rst** and **.rest**:

```
(require 'cl)
(defvar emacs-directory "~/emacs/"
        "The directory containing the emacs configuration files.")
(pushnew (expand-file-name emacs-directory) load-path)
(require 'rst)
(setq auto-mode-alist
       (append '(("\\.rst$" . rst-mode)
                 ("\\.rest$" . rst-mode)) auto-mode-alist))
```

CHAPTER 9

# Part I: Foundations

CHAPTER 10

# Python for Programmers

This book assumes you're an experienced programmer, and it's best if you have learned Python through another book. For everyone else, this chapter gives a programmer's introduction to the language.

This is not an introductory book. I am assuming that you have worked your way through at least *Learning Python* (by Mark Lutz & David Ascher; Oreilly, 1999) or an equivalent text before coming to this book.

This brief introduction is for the experienced programmer (which is what you should be if you're reading this book). You can refer to the full documentation at www.Python.org.

I find the HTML page A Python Quick Reference to be incredibly useful.

In addition, I'll assume you have more than just a grasp of the syntax of Python. You should have a good understanding of objects and what they're about, including polymorphism.

On the other hand, by going through this book you're going to learn a *lot* about object-oriented programming by seeing objects used in many different situations. If your knowledge of objects is rudimentary, it will get much stronger in the process of understanding the designs in this book.

## Scripting vs. Programming

Python is often referred to as a scripting language, but scripting languages tend to be limiting, especially in the scope of the problems that they solve. Python, on the other hand, is a programming language that also supports scripting. It *is* marvelous for scripting, and you may find yourself replacing all your batch files, shell scripts, and simple programs with Python scripts. But it is far more than a scripting language.

The goal of Python is improved productivity. This productivity comes in many ways, but the language is designed to aid you as much as possible, while hindering you as little as possible with arbitrary rules or any requirement that you use a particular set of features. Python is practical; Python language design decisions were based on providing the maximum benefits to the programmer.

Python is very clean to write and especially to read. You will find that it's quite easy to read your own code long after you've written it, and also to read other people's code. This is accomplished partially through clean, to-the-point syntax, but a major factor in code readability is indentation - scoping in Python is determined by indentation. For example:

```
# PythonForProgrammers/if.py
response = "yes"
if response == "yes":
    print("affirmative")
    val = 1
print("continuing...")
```

The '**#**' denotes a comment that goes until the end of the line, just like C++ and Java '**//**' comments.

First notice that the basic syntax of Python is C-ish as you can see in the **if** statement. But in a C **if**, you would be required to use parentheses around the conditional, whereas they are not necessary in Python (it won't complain if you use them anyway).

The conditional clause ends with a colon, and this indicates that what follows will be a group of indented statements, which are the "then" part of the **if** statement. In this case, there is a "print" statement which sends the result to standard output, followed by an assignment to a variable named **val**. The subsequent statement is not indented so it is no longer part of the **if**. Indenting can nest to any level, just like curly braces in C++ or Java, but unlike those languages there is no option (and no argument) about where the braces are placed - the compiler forces everyone's code to be formatted the same way, which is one of the main reasons for Python's consistent readability.

Python normally has only one statement per line (you can put more by separating them with semicolons), thus no terminating semicolon is necessary. Even from the brief example above you can see that the language is designed to be as simple as possible, and yet still very readable.

## Built-In Containers

With languages like C++ and Java, containers are add-on libraries and not integral to the language. In Python, the essential nature of containers for programming is acknowledged by building them into the core of the language: both lists and associative arrays (a.k.a. maps, dictionaries, hash tables) are fundamental data types. This adds much to the elegance of the language.

In addition, the **for** statement automatically iterates through lists rather than just counting through a sequence of numbers. This makes a lot of sense when you think about it, since you're almost always using a **for** loop to step through an array or a container. Python formalizes this by automatically making **for** use an iterator that works through a sequence. Here's an example:

```
# PythonForProgrammers/list.py
list = [ 1, 3, 5, 7, 9, 11 ]
print(list)
list.append(13)
for x in list:
    print(x)
```

The first line creates a list. You can print the list and it will look exactly as you put it in (in contrast, remember that I had to create a special **Arrays2** class in *Thinking in Java* in order to print arrays in Java). Lists are like Java containers - you can add new elements to them (here, **append( )** is used) and they will automatically resize themselves. The **for** statement creates an iterator **x** which takes on each value in the list.

You can create a list of numbers with the **range( )** function, so if you really need to imitate C's **for**, you can.

Notice that there aren't any type declarations - the object names simply appear, and Python infers their type by the way that you use them. It's as if Python is designed so that you only need to press the keys that absolutely must. You'll find after you've worked with Python for a short while that you've been using up a lot of brain cycles parsing semicolons, curly braces, and all sorts of other extra verbiage that was demanded

by your non-Python programming language but didn't actually describe what your program was supposed to do.

## Functions

To create a function in Python, you use the **def** keyword, followed by the function name and argument list, and a colon to begin the function body. Here is the first example turned into a function:

```
# PythonForProgrammers/myFunction.py
def myFunction(response):
    val = 0
    if response == "yes":
        print("affirmative")
        val = 1
    print("continuing...")
    return val

print(myFunction("no"))
print(myFunction("yes"))
```

Notice there is no type information in the function signature - all it specifies is the name of the function and the argument identifiers, but no argument types or return types. Python is a *structurally-typed* language, which means it puts the minimum possible requirements on typing. For example, you could pass and return different types from the same function:

```
# PythonForProgrammers/differentReturns.py
def differentReturns(arg):
    if arg == 1:
        return "one"
    if arg == "one":
        return True

print(differentReturns(1))
print(differentReturns("one"))
```

The only constraints on an object that is passed into the function are that the function can apply its operations to that object, but other than that, it doesn't care. Here, the same function applies the '**+**' operator to integers and strings:

```
# PythonForProgrammers/sum.py
def sum(arg1, arg2):
    return arg1 + arg2

print(sum(42, 47))
print(sum('spam ', "eggs"))
```

When the operator '**+**' is used with strings, it means concatenation (yes, Python supports operator overloading, and it does a nice job of it).

## Strings

The above example also shows a little bit about Python string handling, which is the best of any language I've seen. You can use single or double quotes to represent strings, which is very nice because if you surround a string with double quotes, you can embed single quotes and vice versa:

```
# PythonForProgrammers/strings.py
print("That isn't a horse")
print('You are not a "Viking"')
print("""You're just pounding two
coconut halves together.""")
print('''"Oh no!" He exclaimed.
"It's the blemange!"''')
print(r'c:\python\lib\utils')
```

Note that Python was not named after the snake, but rather the Monty Python comedy troupe, and so examples are virtually required to include Python-esque references.

The triple-quote syntax quotes everything, including newlines. This makes it particularly useful for doing things like generating web pages (Python is an especially good CGI language), since you can just triple-quote the entire page that you want without any other editing.

The '**r**' right before a string means "raw," which takes the backslashes literally so you don't have to put in an extra backslash in order to insert a literal backslash.

Substitution in strings is exceptionally easy, since Python uses C's **printf()** substitution syntax, but for any string at all. You simply follow the string with a '**%**' and the values to substitute:

```
# PythonForProgrammers/stringFormatting.py
val = 47
print("The number is %d" % val)
val2 = 63.4
s = "val: %d, val2: %f" % (val, val2)
print(s)
```

As you can see in the second case, if you have more than one argument you surround them in parentheses (this forms a *tuple*, which is a list that cannot be modified - you can also use regular lists for multiple arguments, but tuples are typical).

All the formatting from **printf()** is available, including control over the number of decimal places and alignment. Python also has very sophisticated regular expressions.

## Classes

Like everything else in Python, the definition of a class uses a minimum of additional syntax. You use the **class** keyword, and inside the body you use **def** to create methods. Here's a simple class:

```
# PythonForProgrammers/SimpleClass.py
class Simple:
    def __init__(self, str):
        print("Inside the Simple constructor")
        self.s = str
    # Two methods:
    def show(self):
        print(self.s)
    def showMsg(self, msg):
        print(msg + ':',
        self.show()) # Calling another method

if __name__ == "__main__":
    # Create an object:
    x = Simple("constructor argument")
```

```
    x.show()
    x.showMsg("A message")
```

Both methods have **self** as their first argument. C++ and Java both have a hidden first argument in their class methods, which points to the object that the method was called for and can be accessed using the keyword **this**. Python methods also use a reference to the current object, but when you are *defining* a method you must explicitly specify the reference as the first argument. Traditionally, the reference is called **self** but you could use any identifier you want (if you do not use **self** you will probably confuse a lot of people, however). If you need to refer to fields in the object or other methods in the object, you must use **self** in the expression. However, when you call a method for an object as in **x.show( )**, you do not hand it the reference to the object - *that* is done for you.

Here, the first method is special, as is any identifier that begins and ends with double underscores. In this case, it defines the constructor, which is automatically called when the object is created, just like in C++ and Java. However, at the bottom of the example you can see that the creation of an object looks just like a function call using the class name. Python's spare syntax makes you realize that the **new** keyword isn't really necessary in C++ or Java, either.

All the code at the bottom is set off by an **if** clause, which checks to see if something called **__name__** is equivalent to **__main__**. Again, the double underscores indicate special names. The reason for the **if** is that any file can also be used as a library module within another program (modules are described shortly). In that case, you just want the classes defined, but you don't want the code at the bottom of the file to be executed. This particular **if** statement is only true when you are running this file directly; that is, if you say on the command line:

```
Python SimpleClass.py
```

However, if this file is imported as a module into another program, the **__main__** code is not executed.

Something that's a little surprising at first is that while in C++ or Java you declare object level fields outside of the methods, you do not declare them in Python. To create an object field, you just name it - using **self** - inside of one of the methods (usually in the constructor, but not always), and space is created when that method is run. This seems a little strange coming from C++ or Java where you must decide ahead of time how much space your object is going to occupy, but it turns out to be a very flexible way to program. If you declare fields using the C++/Java style, they implicitly become class level fields (similar to the static fields in C++/Java)

## Inheritance

Because Python is dynamically typed, it doesn't really care about interfaces - all it cares about is applying operations to objects (in fact, Java's **interface** keyword would be wasted in Python). This means that inheritance in Python is different from inheritance in C++ or Java, where you often inherit simply to establish a common interface. In Python, the only reason you inherit is to inherit an implementation - to re-use the code in the base class.

If you're going to inherit from a class, you must tell Python to bring that class into your new file. Python controls its name spaces as aggressively as Java does, and in a similar fashion (albeit with Python's penchant for simplicity). Every time you create a file, you implicitly create a module (which is like a package in Java) with the same name as that file. Thus, no **package** keyword is needed in Python. When you want to use a module, you just say **import** and give the name of the module. Python searches the PYTHONPATH in the same way that Java searches the CLASSPATH (but for some reason, Python doesn't have the same kinds of pitfalls as Java does) and reads in the file. To refer to any of the functions or classes within a module, you give the module name, a period, and the function or class name. If you don't want the trouble of qualifying the name, you can say

*from module import name(s)*

Where "name(s)" can be a list of names separated by commas.

You inherit a class (or classes - Python supports multiple inheritance) by listing the name(s) of the class inside parentheses after the name of the inheriting class. Note that the **Simple** class, which resides in the file (and thus, module) named **SimpleClass** is brought into this new name space using an **import** statement:

```
# PythonForProgrammers/Simple2.py
from SimpleClass import Simple

class Simple2(Simple):
    def __init__(self, str):
        print("Inside Simple2 constructor")
        # You must explicitly call
        # the base-class constructor:
        Simple.__init__(self, str)
    def display(self):
        self.showMsg("Called from display()")
    # Overriding a base-class method
    def show(self):
        print("Overridden show() method")
        # Calling a base-class method from inside
        # the overridden method:
        Simple.show(self)

class Different:
    def show(self):
        print("Not derived from Simple")

if __name__ == "__main__":
    x = Simple2("Simple2 constructor argument")
    x.display()
    x.show()
    x.showMsg("Inside main")
    def f(obj): obj.show() # One-line definition
    f(x)
    f(Different())
```

---

**Note:** you don't have to explicitly call the base-class constructor if the argument list is the same. Show example.

---

---

**Note:** (Reader) The note above is confusing. Did not understand. IMHO one still needs to invoke the base-class constructor if the argument is the same. Probably one needs to state that in case the base class constructor functionality continues to be adequate for the derived class, then a new constructor need not be declared for the derived class at all.

---

**Simple2** is inherited from **Simple**, and in the constructor, the base-class constructor is called. In **display( )**, **showMsg( )** can be called as a method of **self**, but when calling the base-class version of the method you are overriding, you must fully qualify the name and pass **self** in as the first argument, as shown in the base-class constructor call. This can also be seen in the overridden version of **show( )**.

In **__main__**, you will see (when you run the program) that the base-class constructor is called. You can also see that the **showMsg( )** method is available in the derived class, just as you would expect with inheritance.

The class **Different** also has a method named **show( )**, but this class is not derived from **Simple**. The **f( )**

method defined in **__main__** demonstrates weak typing: all it cares about is that **show( )** can be applied to **obj**, and it doesn't have any other type requirements. You can see that **f( )** can be applied equally to an object of a class derived from **Simple** and one that isn't, without discrimination. If you're a C++ programmer, you should see that the objective of the C++ **template** feature is exactly this: to provide weak typing in a strongly-typed language. Thus, in Python you automatically get the equivalent of templates - without having to learn that particularly difficult syntax and semantics.

## Useful Techniques

- You can turn a list into function arguments using `*`:

```
def f(a,b,c): print a, b, c
x = [1,2,3]
f(*x)
f(*(1,2,3))
```

- You can compose classes using `import`. Here's a method that can be reused by multiple classes:

```
# PythonForProgrammers/utility.py
def f(self): print "utility.f()!!!"
```

Here's how you compose that method into a class:

```
# PythonForProgrammers/compose.py
class Compose:
    from utility import f

Compose().f()
```

- Basic functional programming with `map()` etc.

**Note:** Suggest Further Topics for inclusion in the introductory chapter

## Further Reading

Python Programming FAQ: http://www.python.org/doc/faq/programming/

Python idioms: http://jaynes.colorado.edu/PythonIdioms.html

Python Tips, Tricks and Hacks: http://www.siafoo.net/article/52

Building a Virtual Environment for Running Python 3: http://pypi.python.org/pypi/virtualenv

Excellent Newsfeed Following Python Articles from Everywhere: http://www.planetpython.org/

CHAPTER 11

# Initialization and Cleanup

## Initialization

### Constructor Calls

Automatic base-class constructor calls.

Calling the base-class constructor first, how to do it using super(), why you should always call it first even if it's optional when to call it.

### __new__() vs. __init__()

### Static Fields

An excellent example of the subtleties of initialization is static fields in classes.

::

```
>>> class Foo(object):
...   x = "a"
...
>>> Foo.x
'a'
>>> f = Foo()
>>> f.x
'a'
>>> f2 = Foo()
>>> f2.x
'a'
>>> f2.x = 'b'
>>> f.x
'a'
>>> Foo.x = 'c'
```

```
>>> f.x
'c'
>>> f2.x
'b'
>>> Foo.x = 'd'
>>> f2.x
'b'
>>> f.x
'd'
>>> f3 = Foo()
>>> f3.x
'd'
>>> Foo.x = 'e'
>>> f3.x
'e'
>>> f2.x
'b'
```

If you assign, you get a new one. If it's modifiable, then unless you assign you are working on a singleton. So a typical pattern is:

```
class Foo:
    something = None # Static: visible to all classes
    def f(self, x):
        if not self.something:
            self.something = [] # New local version for this object
        self.something.append(x)
```

This is not a serious example because you would naturally just initialize `something` in `Foo`'s constructor.

## Cleanup

Cleanup happens to globals by setting them to `None` (what about locals?). Does the act of setting them to None cause __del__ to be called, or is __del__ called by Python before a global is set to None?

Consider the following:

```
class Counter:
    Count = 0   # This represents the count of objects of this class
    def __init__(self, name):
        self.name = name
        print name, 'created'
        Counter.Count += 1
    def __del__(self):
        print self.name, 'deleted'
        Counter.Count -= 1
        if Counter.Count == 0:
            print 'Last Counter object deleted'
        else:
            print Counter.Count, 'Counter objects remaining'

x = Counter("First")
del x
```

Without the final del, you get an exception. Shouldn't the normal cleanup process take care of this?

From the Python docs regarding **__del__**:

> Warning: Due to the precarious circumstances under which __del__() methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to sys.stderr instead. Also, when __del__() is invoked in response to a module being deleted (e.g., when execution of the program is done), *other globals referenced by the __del__() method may already have been deleted*. For this reason, __del__() methods should do the absolute minimum needed to maintain external invariants.

Without the explicit call to `del`, `__del__` is only called at the end of the program, Counter and/or Count may have already been GC-ed by the time `__del__` is called (the order in which objects are collected is not deterministic). The exception means that Counter has already been collectd. You can't do anything particularly fancy with __del__.

There are two possible solutions here.

1. Use an explicit finalizer method, such as `close()` for file objects.
2. Use weak references.

Here's an example of weak references, using a WeakValueDictionary and the trick of mapping id(self) to self:

```
from weakref import WeakValueDictionary

class Counter:
    _instances = WeakValueDictionary()
    @property
    def Count(self):
        return len(self._instances)

    def __init__(self, name):
        self.name = name
        self._instances[id(self)] = self
        print name, 'created'

    def __del__(self):
        print self.name, 'deleted'
        if self.Count == 0:
            print 'Last Counter object deleted'
        else:
            print self.Count, 'Counter objects remaining'

x = Counter("First")
```

Now cleanup happens properly without the need for an explicit call to `del`.

## Further Reading

# CHAPTER 12

## Unit Testing & Test-Driven Development

**Note:** This chapter has not had any significant translation yet. Should introduce and compare the various common test systems.

One of the important recent realizations is the dramatic value of unit testing.

This is the process of building integrated tests into all the code that you create, and running those tests every time you do a build. It's as if you are extending the compiler, telling it more about what your program is supposed to do. That way, the build process can check for more than just syntax errors, since you teach it how to check for semantic errors as well.

C-style programming languages, and C++ in particular, have typically valued performance over programming safety. The reason that developing programs in Java is so much faster than in C++ (roughly twice as fast, by most accounts) is because of Java's safety net: features like better type checking, enforced exceptions and garbage collection. By integrating unit testing into your build process, you are extending this safety net, and the result is that you can develop faster. You can also be bolder in the changes that you make, and more easily refactor your code when you discover design or implementation flaws, and in general produce a better product, faster.

Unit testing is not generally considered a design pattern; in fact, it might be considered a "development pattern," but perhaps there are enough "pattern" phrases in the world already. Its effect on development is so significant that it will be used throughout this book, and thus will be introduced here.

My own experience with unit testing began when I realized that every program in a book must be automatically extracted and organized into a source tree, along with appropriate makefiles (or some equivalent technology) so that you could just type **make** to build the whole tree. The effect of this process on the code quality of the book was so immediate and dramatic that it soon became (in my mind) a requisite for any programming book-how can you trust code that you didn't compile? I also discovered that if I wanted to make sweeping changes, I could do so using search-and-replace throughout the book, and also bashing the code around at will. I knew that if I introduced a flaw, the code extractor and the makefiles would flush it out.

As programs became more complex, however, I also found that there was a serious hole in my system. Being able to successfully compile programs is clearly an important first step, and for a published book it seemed

a fairly revolutionary one-usually due to the pressures of publishing, it's quite typical to randomly open a programming book and discover a coding flaw. However, I kept getting messages from readers reporting semantic problems in my code (in *Thinking in Java*). These problems could only be discovered by running the code. Naturally, I understood this and had taken some early faltering steps towards implementing a system that would perform automatic execution tests, but I had succumbed to the pressures of publishing, all the while knowing that there was definitely something wrong with my process and that it would come back to bite me in the form of embarrassing bug reports (in the open source world, embarrassment is one of the prime motivating factors towards increasing the quality of one's code!).

The other problem was that I was lacking a structure for the testing system. Eventually, I started hearing about unit testing and Junit[1], which provided a basis for a testing structure. However, even though JUnit is intended to make the creation of test code easy, I wanted to see if I could make it even easier, applying the Extreme Programming principle of "do the simplest thing that could possibly work" as a starting point, and then evolving the system as usage demands (In addition, I wanted to try to reduce the amount of test code, in an attempt to fit more functionality in less code for screen presentations). This chapter is the result.

## Write Tests First

As I mentioned, one of the problems that I encountered-that most people encounter, it turns out-was submitting to the pressures of publishing and as a result letting tests fall by the wayside. This is easy to do if you forge ahead and write your program code because there's a little voice that tells you that, after all, you've got it working now, and wouldn't it be more interesting/useful/expedient to just go on and write that other part (we can always go back and write the tests later). As a result, the tests take on less importance, as they often do in a development project.

The answer to this problem, which I first found described in *Extreme Programming Explained*, is to write the tests *before* you write the code. This may seem to artificially force testing to the forefront of the development process, but what it actually does is to give testing enough additional value to make it essential. If you write the tests first, you:

1. Describe what the code is supposed to do, not with some external graphical tool but with code that actually lays the specification down in concrete, verifiable terms.
2. Provide an example of how the code should be used; again, this is a working, tested example, normally showing all the important method calls, rather than just an academic description of a library.
3. Provide a way to verify when the code is finished (when all the tests run correctly).

Thus, if you write the tests first then testing becomes a development tool, not just a verification step that can be skipped if you happen to feel comfortable about the code that you just wrote (a comfort, I have found, that is usually wrong).

You can find convincing arguments in *Extreme Programming Explained*, as "write tests first" is a fundamental principle of XP. If you aren't convinced you need to adopt any of the changes suggested by XP, note that according to Software Engineering Institute (SEI) studies, nearly 70% of software organizations are stuck in the first two levels of SEI's scale of sophistication: chaos, and slightly better than chaos. If you change nothing else, add automated testing.

## Simple Python Testing

Sanity check for a quick test of the programs in this book, and to append the output of each program (as a string) to its listing:

[1] *http://www.junit.org*

```
# SanityCheck.py
#! /usr/bin/env python
import string, glob, os
# Do not include the following in the automatic tests:
exclude = ("SanityCheck.py", "BoxObserver.py",)

def visitor(arg, dirname, names):
    dir = os.getcwd()
    os.chdir(dirname)
    try:
        pyprogs = [p for p in glob.glob('*.py') if p not in exclude ]
        if not pyprogs: return
        print('[' + os.getcwd() + ']')
        for program in pyprogs:
            print('\t', program)
            os.system("python %s > tmp" % program)
            file = open(program).read()
            output = open('tmp').read()
            # Append program output if it's not already there:
            if file.find("output = '''") == -1 and len(output) > 0:
                divider = '#' * 50 + '\n'
                file = file.replace('#' + ':~', '#<hr>\n')
                file += "output = '''\n" + open('tmp').read() + "'''\n"
                open(program,'w').write(file)
    finally:
        os.chdir(dir)

if __name__ == "__main__":
    os.path.walk('.', visitor, None)
```

Just run this from the root directory of the code listings for the book; it will descend into each subdirectory and run the program there. An easy way to check things is to redirect standard output to a file, then if there are any errors they will be the only thing that appears at the console during program execution.

## A Very Simple Framework

As mentioned, a primary goal of this code is to make the writing of unit testing code very simple, even simpler than with JUnit. As further needs are discovered *during the use* of this system, then that functionality can be added, but to start with the framework will just provide a way to easily create and run tests, and report failure if something breaks (success will produce no results other than normal output that may occur during the running of the test). My intended use of this framework is in makefiles, and **make** aborts if there is a non- zero return value from the execution of a command. The build process will consist of compilation of the programs and execution of unit tests, and if **make** gets all the way through successfully then the system will be validated, otherwise it will abort at the place of failure. The error messages will report the test that failed but not much else, so that you can provide whatever granularity that you need by writing as many tests as you want, each one covering as much or as little as you find necessary.

In some sense, this framework provides an alternative place for all those "print" statements I've written and later erased over the years.

To create a set of tests, you start by making a **static** inner class inside the class you wish to test (your test code may also test other classes; it's up to you). This test code is distinguished by inheriting from **UnitTest**:

```
# UnitTesting/UnitTest.py
# The basic unit testing class
```

```
class UnitTest:
    testID = ""
    errors = []
    # Override cleanup() if test object creation allocates non-memory
    # resources that must be cleaned up:
    def cleanup(self): pass
    # Verify a condition is true:
    def affirm(condition):
        if(!condition)
            UnitTest.errors.append("failed: " + UnitTest.testID)
```

The only testing method [[ So far ]] is **affirm( )**[2], which is **protected** so that it can be used from the inheriting class. All this method does is verify that something is **true**. If not, it adds an error to the list, reporting that the current test (established by the **static testID**, which is set by the test-running program that you shall see shortly) has failed. Although this is not a lot of information-you might also wish to have the line number, which could be extracted from an exception-it may be enough for most situations.

Unlike JUnit (which uses **setUp( )** and **tearDown( )** methods), test objects will be built using ordinary Python construction. You define the test objects by creating them as ordinary class members of the test class, and a new test class object will be created for each test method (thus preventing any problems that might occur from side effects between tests). Occasionally, the creation of a test object will allocate non-memory resources, in which case you must override **cleanup( )** to release those resources.

## Writing Tests

Writing tests becomes very simple. Here's an example that creates the necessary **static** inner class and performs trivial tests:

```
# UnitTesting/TestDemo.py
# Creating a test

class TestDemo:
    objCounter = 0
    id = ++objCounter
    def TestDemo(String s):
        print(s + ": count = " + id)

    def close(self):
        print("Cleaning up: " + id)

    def someCondition(self): return True
    class Test(UnitTest):
        TestDemo test1 = TestDemo("test1")
        TestDemo test2 = TestDemo("test2")
        def cleanup(self):
            test2.close()
            test1.close()

        def testA(self):
            print("TestDemo.testA")
            affirm(test1.someCondition())

        def testB(self):
```

[2] I had originally called this **assert()**, but that word became reserved in JDK 1.4 when assertions were added to the language.

```
            print("TestDemo.testB")
            affirm(test2.someCondition())
            affirm(TestDemo.objCounter != 0)

        # Causes the build to halt:
        #! def test3(): affirm(0)
```

The **test3( )** method is commented out because, as you'll see, it causes the automatic build of this book's source-code tree to stop.

You can name your inner class anything you'd like; the only important factor is that it **extends UnitTest**. You can also include any necessary support code in other methods. Only **public** methods that take no arguments and return **void** will be treated as tests (the names of these methods are also not constrained).

The above test class creates two instances of **TestDemo**. The **TestDemo** constructor prints something, so that we can see it being called. You could also define a default constructor (the only kind that is used by the test framework), although none is necessary here. The **TestDemo** class has a **close( )** method which suggests it is used as part of object cleanup, so this is called in the overridden **cleanup( )** method in **Test**.

The testing methods use the **affirm( )** method to validate expressions, and if there is a failure the information is stored and printed after all the tests are run. Of course, the **affirm( )** arguments are usually more complicated than this; you'll see more examples throughout the rest of this book.

Notice that in **testB( )**, the **private** field **objCounter** is accessible to the testing code-this is because **Test** has the permissions of an inner class.

You can see that writing test code requires very little extra effort, and no knowledge other than that used for writing ordinary classes.

To run the tests, you use **RunUnitTests.py** (which will be introduced shortly). The command for the above code looks like this:

**java com.bruceeckel.test.RunUnitTests TestDemo**

It produces the following output:

```
test1: count = 1
test2: count = 2
TestDemo.testA
Cleaning up: 2
Cleaning up: 1
test1: count = 3
test2: count = 4
TestDemo.testB
Cleaning up: 4
Cleaning up: 3
```

All the output is noise as far as the success or failure of the unit testing is concerned. Only if one or more of the unit tests fail does the program returns a non-zero value to terminate the **make** process after the error messages are produced. Thus, you can choose to produce output or not, as it suits your needs, and the test class becomes a good place to put any printing code you might need- if you do this, you tend to keep such code around rather than putting it in and stripping it out as is typically done with tracing code.

If you need to add a test to a class derived from one that already has a test class, it's no problem, as you can see here:

```
# UnitTesting/TestDemo2.py
# Inheriting from a class that
# already has a test is no problem.
```

```
class TestDemo2(TestDemo):
    def __init__(self, s): TestDemo.__init__(s)
    # You can even use the same name
    # as the test class in the base class:
    class Test(UnitTest):
        def testA(self):
            print("TestDemo2.testA")
            affirm(1 + 1 == 2)

        def testB(self):
            print("TestDemo2.testB")
            affirm(2 * 2 == 4)
```

Even the name of the inner class can be the same. In the above code, all the assertions are always true so the tests will never fail.

## White-Box & Black-Box Tests

The unit test examples so far are what are traditionally called *white-box tests*. This means that the test code has complete access to the internals of the class that's being tested (so it might be more appropriately called "transparent box" testing). White-box testing happens automatically when you make the unit test class as an inner class of the class being tested, since inner classes automatically have access to all their outer class elements, even those that are **private**.

A possibly more common form of testing is *black-box testing*, which refers to treating the class under test as an impenetrable box. You can't see the internals; you can only access the **public** portions of the class. Thus, black-box testing corresponds more closely to functional testing, to verify the methods that the client programmer is going to use. In addition, black-box testing provides a minimal instruction sheet to the client programmer - in the absence of all other documentation, the black-box tests at least demonstrate how to make basic calls to the **public** class methods.

To perform black-box tests using the unit-testing framework presented in this book, all you need to do is create your test class as a global class instead of an inner class. All the other rules are the same (for example, the unit test class must be **public**, and derived from **UnitTest**).

There's one other caveat, which will also provide a little review of Java packages. If you want to be completely rigorous, you must put your black-box test class in a separate directory than the class it tests, otherwise it will have package access to the elements of the class being tested. That is, you'll be able to access **protected** and **friendly** elements of the class being tested. Here's an example:

```
# UnitTesting/Testable.py

class Testable:
    def f1(): pass
    def f2(self): pass # "Friendly": package access
    def f3(self): pass # Also package access
    def f4(self): pass
```

Normally, the only method that should be directly accessible to the client programmer is **f4( )**. However, if you put your black-box test in the same directory, it automatically becomes part of the same package (in this case, the default package since none is specified) and then has inappropriate access:

```
# UnitTesting/TooMuchAccess.py

class TooMuchAccess(UnitTest):
```

```
    Testable tst = Testable()
    def test1(self):
        tst.f2() # Oops!
        tst.f3() # Oops!
        tst.f4() # OK
```

You can solve the problem by moving **TooMuchAccess.py** into its own subdirectory, thereby putting it in its own default package (thus a different package from **Testable.py**). Of course, when you do this, then **Testable** must be in its own package, so that it can be imported (note that it is also possible to import a "package-less" class by giving the class name in the **import** statement and ensuring that the class is in your CLASSPATH):

```
# UnitTesting/testable/Testable.py
package c02.testable

class Testable:
    def f1(): pass
    def f2(self): # "Friendly": package access
    def f3(self): # Also package access
    def f4(self):
```

Here's the black-box test in its own package, showing how only public methods may be called:

```
# UnitTesting/BlackBoxTest.py

class BlackBoxTest(UnitTest):
    Testable tst = Testable()
    def test1(self):
        #! tst.f2() # Nope!
        #! tst.f3() # Nope!
        tst.f4() # Only public methods available
```

Note that the above program is indeed very similar to the one that the client programmer would write to use your class, including the imports and available methods. So it does make a good programming example. Of course, it's easier from a coding standpoint to just make an inner class, and unless you're ardent about the need for specific black-box testing you may just want to go ahead and use the inner classes (with the knowledge that if you need to you can later extract the inner classes into separate black-box test classes, without too much effort).

## Running tests

The program that runs the tests makes significant use of reflection so that writing the tests can be simple for the client programmer:

```
# UnitTesting/RunUnitTests.py
# Discovering the unit test
# class and running each test.

class RunUnitTests:
    def require(requirement, errmsg):
        if(!requirement):
            print(errmsg)
            sys.exit()

    def main(self, args):
```

```
        require(args.length == 1,
          "Usage: RunUnitTests qualified-class")
        try:
            Class c = Class.forName(args[0])
            # Only finds the inner classes
            # declared in the current class:
            Class[] classes = c.getDeclaredClasses()
            Class ut = null
            for(int j = 0 j < classes.length j++):
                # Skip inner classes that are
                # not derived from UnitTest:
                if(!UnitTest.class.
                    isAssignableFrom(classes[j]))
                    continue
                ut = classes[j]
                break # Finds the first test class only

            # If it found an inner class,
            # that class must be static:
            if(ut != null)
                require(
                  Modifier.isStatic(ut.getModifiers()),
                  "inner UnitTest class must be static")
            # If it couldn't find the inner class,
            # maybe it's a regular class (for black-
            # box testing:
            if(ut == null)
                if(UnitTest.class.isAssignableFrom(c))
                    ut = c
            require(ut != null,
              "No UnitTest class found")
            require(
              Modifier.isPublic(ut.getModifiers()),
              "UnitTest class must be public")
            Method[] methods = ut.getDeclaredMethods()
            for(int k = 0 k < methods.length k++):
                Method m = methods[k]
                # Ignore overridden UnitTest methods:
                if(m.getName().equals("cleanup"))
                    continue
                # Only public methods with no
                # arguments and void return
                # types will be used as test code:
                if(m.getParameterTypes().length == 0 &&
                   m.getReturnType() == void.class &&
                   Modifier.isPublic(m.getModifiers())):
                    # The name of the test is
                    # used in error messages:
                    UnitTest.testID = m.getName()
                    # A instance of the
                    # test object is created and
                    # cleaned up for each test:
                    Object test = ut.newInstance()
                    m.invoke(test, Object[0])
                    ((UnitTest)test).cleanup()

        except e:
            e.printStackTrace(System.err)
```

```
            # Any exception will return a nonzero
            # value to the console, so that
            # 'make' will abort:
            System.err.println("Aborting make")
            System.exit(1)

        # After all tests in this class are run,
        # display any results. If there were errors,
        # abort 'make' by returning a nonzero value.
        if(UnitTest.errors.size() != 0):
            it = UnitTest.errors.iterator()
            while(it.hasNext()):
                print(it.next())
            sys.exit(1)
```

## Automatically Executing Tests

## Exercises

1. Install this book's source code tree and ensure that you have a **make** utility installed on your system (Gnu **make** is freely available on the internet at various locations). In **TestDemo.py**, un-comment **test3( )**, then type **make** and observe the results.
2. Modify TestDemo.py by adding a new test that throws an exception. Type **make** and observe the results.
3. Modify your solutions to the exercises in Chapter 1 by adding unit tests. Write makefiles that incorporate the unit tests.

# CHAPTER 13

# Python 3 Language Changes

Covers language features that don't require their own chapters.

**Note:** If a section in this chapter grows too large it may require its own chapter.

CHAPTER 14

# Decorators

**Note:** This chapter is a work in progress; it's probably better if you don't begin making changes until I've finished the original version, which is being posted as a series on my weblog.

This amazing feature appeared in the language almost apologetically and with concern that it might not be that useful.

I predict that in time it will be seen as one of the more powerful features in the language. The problem is that all the introductions to decorators that I have seen have been rather confusing, so I will try to rectify that here.

## Decorators vs. the Decorator Pattern

First, you need to understand that the word "decorator" was used with some trepidation in Python, because there was concern that it would be completely confused with the *Decorator* pattern from the Design Patterns book. At one point other terms were considered for the feature, but "decorator" seems to be the one that sticks.

Indeed, you can use Python decorators to implement the *Decorator* pattern, but that's an extremely limited use of it. Python decorators, I think, are best equated to macros.

## History of Macros

The macro has a long history, but most people will probably have had experience with C preprocessor macros. The problems with C macros were (1) they were in a different language (not C) and (2) the behavior was sometimes bizarre, and often inconsistent with the behavior of the rest of C.

Both Java and C# have added *annotations*, which allow you to do some things to elements of the language. Both of these have the problems that (1) to do what you want, you sometimes have to jump through some

enormous and untenable hoops, which follows from (2) these annotation features have their hands tied by the bondage-and-discipline (or as Martin Fowler gently puts it: "Directing") nature of those languages.

In a slightly different vein, many C++ programmers (myself included) have noted the generative abilities of C++ templates and have used that feature in a macro- like fashion.

Many other languages have incorporated macros, but without knowing much about it I will go out on a limb and say that Python decorators are similar to Lisp macros in power and possibility.

## The Goal of Macros

I think it's safe to say that the goal of macros in a language is to provide a way to modify elements of the language. That's what decorators do in Python – they modify functions, and in the case of *class decorators*, entire classes. This is why they usually provide a simpler alternative to metaclasses.

The major failings of most language's self-modification approaches are that they are too restrictive and that they require a different language (I'm going to say that Java annotations with all the hoops you must jump through to produce an interesting annotation comprises a "different language").

Python falls into Fowler's category of "enabling" languages, so if you want to do modifications, why create a different or restricted language? Why not just use Python itself? And that's what Python decorators do.

## What Can You Do With Decorators?

Decorators allow you to inject or modify code in functions or classes. Sounds a bit like *Aspect-Oriented Programming* (AOP) in Java, doesn't it? Except that it's both much simpler and (as a result) much more powerful. For example, suppose you'd like to do something at the entry and exit points of a function (such as perform some kind of security, tracing, locking, etc. – all the standard arguments for AOP). With decorators, it looks like this:

```
@entryExit
def func1():
    print("inside func1()")

@entryExit
def func2():
    print("inside func2()")
```

The `@` indicates the application of the decorator.

## Function Decorators

A function decorator is applied to a function definition by placing it on the line before that function definition begins. For example:

```
@myDecorator
def aFunction():
    print("inside aFunction")
```

When the compiler passes over this code, `aFunction()` is compiled and the resulting function object is passed to the `myDecorator` code, which does something to produce a function-like object that is then substituted for the original `aFunction()`.

What does the `myDecorator` code look like? Well, most introductory examples show this as a function, but I've found that it's easier to start understanding decorators by using classes as decoration mechanisms instead of functions. In addition, it's more powerful.

The only constraint upon the object returned by the decorator is that it can be used as a function – which basically means it must be callable. Thus, any classes we use as decorators must implement `__call__`.

What should the decorator do? Well, it can do anything but usually you expect the original function code to be used at some point. This is not required, however:

```
# PythonDecorators/my_decorator.py
class my_decorator(object):

    def __init__(self, f):
        print("inside my_decorator.__init__()")
        f() # Prove that function definition has completed

    def __call__(self):
        print("inside my_decorator.__call__()")

@my_decorator
def aFunction():
    print("inside aFunction()")

print("Finished decorating aFunction()")

aFunction()
```

When you run this code, you see:

```
inside my_decorator.__init__()
inside aFunction()
Finished decorating aFunction()
inside my_decorator.__call__()
```

Notice that the constructor for `my_decorator` is executed at the point of decoration of the function. Since we can call `f()` inside `__init__()`, it shows that the creation of `f()` is complete before the decorator is called. Note also that the decorator constructor receives the function object being decorated. Typically, you'll capture the function object in the constructor and later use it in the `__call__()` method (the fact that decoration and calling are two clear phases when using classes is why I argue that it's easier and more powerful this way).

When `aFunction()` is called after it has been decorated, we get completely different behavior; the `my_decorator.__call__()` method is called instead of the original code. That's because the act of decoration *replaces* the original function object with the result of the decoration – in our case, the `my_decorator` object replaces `aFunction`. Indeed, before decorators were added you had to do something much less elegant to achieve the same thing:

```
def foo(): pass
foo = staticmethod(foo)
```

With the addition of the `@` decoration operator, you now get the same result by saying:

```
@staticmethod
def foo(): pass
```

This is the reason why people argued against decorators, because the `@` is just a little syntax sugar meaning "pass a function object through another function and assign the result to the original function."

The reason I think decorators will have such a big impact is because this little bit of syntax sugar changes the way you think about programming. Indeed, it brings the idea of "applying code to other code" (i.e.: macros) into mainstream thinking by formalizing it as a language construct.

## Slightly More Useful

Now let's go back and implement the first example. Here, we'll do the more typical thing and actually use the code in the decorated functions:

```
# PythonDecorators/entry_exit_class.py
class entry_exit(object):

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Entering", self.f.__name__)
        self.f()
        print("Exited", self.f.__name__)

@entry_exit
def func1():
    print("inside func1()")

@entry_exit
def func2():
    print("inside func2()")

func1()
func2()
```

The output is:

```
Entering func1
inside func1()
Exited func1
Entering func2
inside func2()
Exited func2
```

You can see that the decorated functions now have the "Entering" and "Exited" trace statements around the call.

The constructor stores the argument, which is the function object. In the call, we use the `__name__` attribute of the function to display that function's name, then call the function itself.

## Using Functions as Decorators

The only constraint on the result of a decorator is that it be callable, so it can properly replace the decorated function. In the above examples, I've replaced the original function with an object of a class that has a `__call__()` method. But a function object is also callable, so we can rewrite the previous example using a function instead of a class, like this:

```
# PythonDecorators/entry_exit_function.py
def entry_exit(f):
    def new_f():
        print("Entering", f.__name__)
        f()
        print("Exited", f.__name__)
    return new_f

@entry_exit
def func1():
    print("inside func1()")

@entry_exit
def func2():
    print("inside func2()")

func1()
func2()
print(func1.__name__)
```

`new_f()` is defined within the body of `entry_exit()`, so it is created and returned when `entry_exit()` is called. Note that `new_f()` is a *closure*, because it captures the actual value of `f`.

Once `new_f()` has been defined, it is returned from `entry_exit()` so that the decorator mechanism can assign the result as the decorated function.

The output of the line `print(func1.__name__)` is `new_f`, because the `new_f` function has been substituted for the original function during decoration. If this is a problem you can change the name of the decorator function before you return it:

```
def entry_exit(f):
    def new_f():
        print("Entering", f.__name__)
        f()
        print("Exited", f.__name__)
    new_f.__name__ = f.__name__
    return new_f
```

The information you can dynamically get about functions, and the modifications you can make to those functions, are quite powerful in Python.

## Review: Decorators without Arguments

If we create a decorator without arguments, the function to be decorated is passed to the constructor, and the `__call__()` method is called whenever the decorated function is invoked:

```
# PythonDecorators/decorator_without_arguments.py
class decorator_without_arguments(object):

    def __init__(self, f):
        """
        If there are no decorator arguments, the function
        to be decorated is passed to the constructor.
        """
        print("Inside __init__()")
        self.f = f
```

```
    def __call__(self, *args):
        """
        The __call__ method is not called until the
        decorated function is called.
        """
        print("Inside __call__()")
        self.f(*args)
        print("After self.f(*args)")

@decorator_without_arguments
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")
sayHello("say", "hello", "argument", "list")
print("After first sayHello() call")
sayHello("a", "different", "set of", "arguments")
print("After second sayHello() call")
```

Any arguments for the decorated function are just passed to `__call__()`. The output is:

```
Inside __init__()
After decoration
Preparing to call sayHello()
Inside __call__()
sayHello arguments: say hello argument list
After self.f(*args)
After first sayHello() call
Inside __call__()
sayHello arguments: a different set of arguments
After self.f(*args)
After second sayHello() call
```

Notice that `__init__()` is the only method called to perform decoration, and `__call__()` is called every time you call the decorated `sayHello()`.

## Decorators with Arguments

The decorator mechanism behaves quite differently when you pass arguments to the decorator.

Let's modify the above example to see what happens when we add arguments to the decorator:

```
# PythonDecorators/decorator_with_arguments.py
class decorator_with_arguments(object):

    def __init__(self, arg1, arg2, arg3):
        """
        If there are decorator arguments, the function
        to be decorated is not passed to the constructor!
        """
        print("Inside __init__()")
        self.arg1 = arg1
        self.arg2 = arg2
```

```
        self.arg3 = arg3

    def __call__(self, f):
        """
        If there are decorator arguments, __call__() is only called
        once, as part of the decoration process! You can only give
        it a single argument, which is the function object.
        """
        print("Inside __call__()")
        def wrapped_f(*args):
            print("Inside wrapped_f()")
            print("Decorator arguments:", self.arg1, self.arg2, self.arg3)
            f(*args)
            print("After f(*args)")
        return wrapped_f

@decorator_with_arguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")
sayHello("say", "hello", "argument", "list")
print("after first sayHello() call")
sayHello("a", "different", "set of", "arguments")
print("after second sayHello() call")
```

From the output, we can see that the behavior changes quite significantly:

```
Inside __init__()
Inside __call__()
After decoration
Preparing to call sayHello()
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: say hello argument list
After f(*args)
after first sayHello() call
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: a different set of arguments
After f(*args)
after second sayHello() call
```

Now the process of decoration calls the constructor and then immediately invokes `__call__()`, which can only take a single argument (the function object) and must return the decorated function object that replaces the original. Notice that `__call__()` is now only invoked once, during decoration, and after that the decorated function that you return from `__call__()` is used for the actual calls.

Although this behavior makes sense – the constructor is now used to capture the decorator arguments, but the object `__call__()` can no longer be used as the decorated function call, so you must instead use `__call__()` to perform the decoration – it is nonetheless surprising the first time you see it because it's acting so much differently than the no-argument case, and you must code the decorator very differently from the no-argument case.

## Decorator Functions with Decorator Arguments

Finally, let's look at the more complex decorator function implementation, where you have to do everything all at once:

```
# PythonDecorators/decorator_function_with_arguments.py
def decorator_function_with_arguments(arg1, arg2, arg3):
    def wrap(f):
        print("Inside wrap()")
        def wrapped_f(*args):
            print("Inside wrapped_f()")
            print("Decorator arguments:", arg1, arg2, arg3)
            f(*args)
            print("After f(*args)")
        return wrapped_f
    return wrap

@decorator_function_with_arguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")
sayHello("say", "hello", "argument", "list")
print("after first sayHello() call")
sayHello("a", "different", "set of", "arguments")
print("after second sayHello() call")
```

Here's the output:

```
Inside wrap()
After decoration
Preparing to call sayHello()
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: say hello argument list
After f(*args)
after first sayHello() call
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: a different set of arguments
After f(*args)
after second sayHello() call
```

The return value of the decorator function must be a function used to wrap the function to be decorated. That is, Python will take the returned function and call it at decoration time, passing the function to be decorated. That's why we have three levels of functions; the inner one is the actual replacement function.

Because of closures, `wrapped_f()` has access to the decorator arguments `arg1`, `arg2` and `arg3`, *without* having to explicitly store them as in the class version. However, this is a case where I find "explicit is better than implicit," so even though the function version is more succinct I find the class version easier to understand and thus to modify and maintain.

## Further Reading

**http://wiki.python.org/moin/PythonDecoratorLibrary** More examples of decorators. Note the number of these examples that use classes rather than functions as decorators.

**http://scratch.tplus1.com/decoratortalk** Matt Wilson's *Decorators Are Fun*.

**http://loveandtheft.org/2008/09/22/python-decorators-explained** Another introduction to decorators.

http://www.siafoo.net/article/68

**http://www.ddj.com/web-development/184406073** Philip Eby introduces decorators.

**http://www.informit.com/articles/article.aspx?p=1309289&seqNum=4** Class Decorators.

**http://www.phyast.pitt.edu/~micheles/python/documentation.html** Michele Simionato's decorator module wraps functions for you. The page includes an introduction and some examples.

**http://www.blueskyonmars.com/projects/paver/** Kevin Djangoor's replacement for `make`; heavy use of decorators.

**http://blog.doughellmann.com/2009/01/converting-from-make-to-paver.html** Doug Hellman describes the experience of converting from make to paver.

**http://www.informit.com/articles/article.aspx?p=1309289&seqNum=4** Class decorators

CHAPTER 15

# Metaprogramming

**Note:** This chapter is written using Python 2.6 syntax; it will be converted to Python 3 at a later date.

Objects are created by other objects: special objects called "classes" that we can set up to spit out objects that are configured to our liking.

Classes are just objects, and they can be modified the same way:

```
>>> class Foo: pass
...
>>> Foo.field = 42
>>> x = Foo()
>>> x.field
42
>>> Foo.field2 = 99
>>> x.field2
99
>>> Foo.method = lambda self: "Hi!"
>>> x.method()
'Hi!'
```

To modify a class, you perform operations on it like any other object. You can add and subtract fields and methods, for example. The difference is that any change you make to a class affects all the objects of that class, even the ones that have already been instantiated.

What creates these special "class" objects? Other special objects, called metaclasses.

The default metaclass is called `type` and in the vast majority of cases it does the right thing. In some situations, however, you can gain leverage by modifying the way that classes are produced – typically by performing extra actions or injecting code. When this is the case, you can use *metaclass programming* to modify the way that some of your class objects are created.

It's worth re-emphasizing that in *the vast majority of cases, you don't need metaclasses*, because it's a fascinating toy and the temptation to use it everywhere can be overwhelming. Some of the examples in this chapter

will show both metaclass and non-metaclass solutions to a problem, so you can see that there's usually another (often simpler) approach.

Some of the functionality that was previously only available with metaclasses is now available in a simpler form using class decorators. It is still useful, however, to understand metaclasses, and certain results can still be achieved only through metaclass programming.

## Basic Metaprogramming

So metaclasses create classes, and classes create instances. Normally when we write a class, the default metaclass `type` is automatically invoked to create that class, and we aren't even aware that it's happening.

It's possible to explicitly code the metaclass' creation of a class. `type` called with one argument produces the type information of an existing class; `type` called with three arguments creates a new class object. The arguments when invoking `type` are the name of the class, a list of base classes, and a dictionary giving the namespace for the class (all the fields and methods). So the equivalent of:

```
class C: pass
```

is:

```
C = type('C', (), {})
```

Classes are often referred to as "types," so this reads fairly sensibly: you're calling a function that creates a new type based on its arguments.

We can also add base classes, fields and methods:

```
# Metaprogramming/MyList.py

def howdy(self, you):
    print("Howdy, " + you)

MyList = type('MyList', (list,), dict(x=42, howdy=howdy))

ml = MyList()
ml.append("Camembert")
print(ml)
print(ml.x)
ml.howdy("John")

print(ml.__class__.__class__)

""" Output:
['Camembert']
42
Howdy, John
"""
```

Note that printing the class of the class produces the metaclass.

The ability to generate classes programmatically using `type` opens up some interesting possibilities. Consider the GreenHouseLanguage.py example in the Jython chapter – all the subclasses in that case were written using repetetive code. We can automate the generation of the subclasses using `type`:

```
# Metaprogramming/GreenHouse.py
```

```
class Event(object):
    events = [] # static

    def __init__(self, action, time):
        self.action = action
        self.time = time
        Event.events.append(self)

    def __cmp__ (self, other):
        "So sort() will compare only on time."
        return cmp(self.time, other.time)

    def run(self):
        print("%.2f: %s" % (self.time, self.action))

    @staticmethod
    def run_events():
        Event.events.sort();
        for e in Event.events:
            e.run()

def create_mc(description):
    "Create subclass using the 'type' metaclass"
    class_name = "".join(x.capitalize() for x in description.split())
    def __init__(self, time):
        Event.__init__(self, description + " [mc]", time)
    globals()[class_name] = \
        type(class_name, (Event,), dict(__init__ = __init__))

def create_exec(description):
    "Create subclass by exec-ing a string"
    class_name = "".join(x.capitalize() for x in description.split())
    klass = """
class %s(Event):
    def __init__(self, time):
        Event.__init__(self, "%s [exec]", time)
""" % (class_name, description)
    exec klass in globals()

if __name__ == "__main__":
    descriptions = ["Light on", "Light off", "Water on", "Water off",
                    "Thermostat night", "Thermostat day", "Ring bell"]
    initializations = "ThermostatNight(5.00); LightOff(2.00); \
        WaterOn(3.30); WaterOff(4.45); LightOn(1.00); \
        RingBell(7.00); ThermostatDay(6.00)"
    [create_mc(dsc) for dsc in descriptions]
    exec initializations in globals()
    [create_exec(dsc) for dsc in descriptions]
    exec initializations in globals()
    Event.run_events()

""" Output:
1.00: Light on [mc]
1.00: Light on [exec]
2.00: Light off [mc]
2.00: Light off [exec]
3.30: Water on [mc]
3.30: Water on [exec]
```

```
4.45: Water off [mc]
4.45: Water off [exec]
5.00: Thermostat night [mc]
5.00: Thermostat night [exec]
6.00: Thermostat day [mc]
6.00: Thermostat day [exec]
7.00: Ring bell [mc]
7.00: Ring bell [exec]
"""
```

The `Event` base class is the same. The classes are created automatically using the `create_mc()` function, which takes its `description` argument and generates a class name from it. Then it defines an `__init__()` method, which it puts into the namespace dictionary for the `type` call, producing a new subclass of `Event`. Note that the resulting class must be inserted into the global namespace, otherwise it will not be seen.

This approach works fine, but then consider the subsequent `create_exec()` function, which accomplishes the same thing by calling `exec` on a string defining the class. This will be much easier to understand by the vast majority of the people reading your code: those who do not understand metaclasses.

## The Metaclass Hook

So far, we've only used the `type` metaclass directly. Metaclass programming involves hooking our own operations into the creation of class objects. This is accomplished by:

1. Writing a subclass of the metaclass `type`.
2. Inserting the new metaclass into the class creation process using the *metaclass hook*.

In Python 2.x, the metaclass hook is a static field in the class called `__metaclass__`. In the ordinary case, this is not assigned so Python just uses `type` to create the class. But if you define `__metaclass__` to point to a callable, Python will call `__metaclass__()` after the initial creation of the class object, passing in the class object, the class name, the list of base classes and the namespace dictionary.

Python 2.x also allows you to assign to the global `__metaclass__` hook, which will be used if there is not a class-local `__metaclass__` hook (is there an equivalent in Python 3?).

Thus, the basic process of metaclass programming looks like this:

```
# Metaprogramming/SimpleMeta1.py
# Two-step metaclass creation in Python 2.x

class SimpleMeta1(type):
    def __init__(cls, name, bases, nmspc):
        super(SimpleMeta1, cls).__init__(name, bases, nmspc)
        cls.uses_metaclass = lambda self : "Yes!"

class Simple1(object):
    __metaclass__ = SimpleMeta1
    def foo(self): pass
    @staticmethod
    def bar(): pass

simple = Simple1()
print([m for m in dir(simple) if not m.startswith('__')])
# A new method has been injected by the metaclass:
print simple.uses_metaclass()
```

```
""" Output:
['bar', 'foo', 'uses_metaclass']
Yes!
"""
```

By convention, when defining metaclasses `cls` is used rather than `self` as the first argument to all methods except `__new__()` (which uses `mcl`, for reasons explained later). `cls` is the class object that is being modified.

Note that the practice of calling the base-class constructor first (via `super()`) in the derived-class constructor should be followed with metaclasses as well.

`__metaclass__` only needs to be callable, so in Python 2.x it's possible to define `__metaclass__` inline:

```
# Metaprogramming/SimpleMeta2.py
# Combining the steps for metaclass creation in Python 2.x

class Simple2(object):
    class __metaclass__(type):
        def __init__(cls, name, bases, nmspc):
            # This won't work:
            # super(__metaclass__, cls).__init__(name, bases, nmspc)
            # Less-flexible specific call:
            type.__init__(cls, name, bases, nmspc)
            cls.uses_metaclass = lambda self : "Yes!"

class Simple3(Simple2): pass
simple = Simple3()
print simple.uses_metaclass()

""" Output:
Yes!
"""
```

The compiler won't accept the `super()` call because it says `__metaclass__` hasn't been defined, forcing us to use the specific call to `type.__init__()`.

Because it only needs to be callable, it's even possible to define `__metaclass__` as a function:

```
# Metaprogramming/SimpleMeta3.py
# A function for __metaclass__ in Python 2.x

class Simple4(object):
    def __metaclass__(name, bases, nmspc):
        cls = type(name, bases, nmspc)
        cls.uses_metaclass = lambda self : "Yes!"
        return cls

simple = Simple4()
print simple.uses_metaclass()

""" Output:
Yes!
"""
```

As you'll see, Python 3 doesn't allow the syntax of these last two examples. Even so, the above example makes it quite clear what's happening: the class object is created, then modified, then returned.

**Note:** Or does it allow that syntax?

### The Metaclass Hook in Python 3

Python 3 changes the metaclass hook. It doesn't disallow the `__metaclass__` field, but it ignores it. Instead, you use a keyword argument in the base-class list:

```
class Simple1(object, metaclass = SimpleMeta1):
    ...
```

This means that none of the (clever) alternative ways of defining `__metaclass__` directly as a class or function are available in Python 3 [[check this]]. All metaclasses must be defined as separate classes. This is probably just as well, as it makes metaclass programs more consistent and thus easier to read and understand.

## Example: Self-Registration of Subclasses

It is sometimes convienient to use inheritance as an organizing mechanism – each sublclass becomes an element of a group that you work on. For example, in **CodeManager.py** in the **Comprehensions** chapter, the subclasses of **Language** were all the languages that needed to be processed. Each **Language** subclass described specific processing traits for that language.

To solve this problem, consider a system that automatically keeps a list of all of its "leaf" subclasses (only the classes that have no inheritors). This way we can easily enumerate through all the subtypes:

```
# Metaprogramming/RegisterLeafClasses.py

class RegisterLeafClasses(type):
    def __init__(cls, name, bases, nmspc):
        super(RegisterLeafClasses, cls).__init__(name, bases, nmspc)
        if not hasattr(cls, 'registry'):
            cls.registry = set()
        cls.registry.add(cls)
        cls.registry -= set(bases) # Remove base classes
    # Metamethods, called on class objects:
    def __iter__(cls):
        return iter(cls.registry)
    def __str__(cls):
        if cls in cls.registry:
            return cls.__name__
        return cls.__name__ + ": " + ", ".join([sc.__name__ for sc in cls])

class Color(object):
    __metaclass__ = RegisterLeafClasses

class Blue(Color): pass
class Red(Color): pass
class Green(Color): pass
class Yellow(Color): pass
print(Color)
class PhthaloBlue(Blue): pass
class CeruleanBlue(Blue): pass
print(Color)
```

```
for c in Color: # Iterate over subclasses
    print(c)

class Shape(object):
    __metaclass__ = RegisterLeafClasses

class Round(Shape): pass
class Square(Shape): pass
class Triangular(Shape): pass
class Boxy(Shape): pass
print(Shape)
class Circle(Round): pass
class Ellipse(Round): pass
print(Shape)

""" Output:
Color: Red, Blue, Green, Yellow
Color: Red, CeruleanBlue, Green, PhthaloBlue, Yellow
Red
CeruleanBlue
Green
PhthaloBlue
Yellow
Shape: Square, Round, Boxy, Triangular
Shape: Square, Ellipse, Circle, Boxy, Triangular
"""
```

Two separate tests are used to show that the registries are independent of each other. Each test shows what happens when another level of leaf classes are added – the former leaf becomes a base class, and so is removed from the registry.

This also introduces *metamethods*, which are defined in the metaclass so that they become methods of the class. That is, you call them on the class rather than object instances, and their first argument is the class object rather than `self`.

### Using Class Decorators

### Using the inspect module

(As in the Comprehensions chapter)

## Example: Making a Class "Final"

It is sometimes convenient to prevent a class from being inherited:

```
# Metaprogramming/Final.py
# Emulating Java's 'final'

class final(type):
    def __init__(cls, name, bases, namespace):
        super(final, cls).__init__(name, bases, namespace)
        for klass in bases:
            if isinstance(klass, final):
                raise TypeError(str(klass.__name__) + " is final")
```

```
class A(object):
    pass

class B(A):
    __metaclass__= final

print B.__bases__
print isinstance(B, final)

# Produces compile-time error:
class C(B):
    pass

""" Output:
(<class '__main__.A'>,)
True
...
TypeError: B is final
"""
```

During class object creation, we check to see if any of the bases are derived from `final`. Notice that using a metaclass makes the new type an instance of that metaclass, even though the metaclass doesn't show up in the base-class list.

Because this process of checking for finality must be installed to happen as the subclasses are created, rather than afterwards as performed by class decorators, it appears that this is an example of something that requires metaclasses and can't be accomplished with class decorators.

## Using `__init__` vs. `__new__` in Metaclasses

It can be confusing when you see metaclass examples that appear to arbitrarily use `__new__` or `__init__` – why choose one over the other?

`__new__` is called for the creation of a new class, while `__init__` is called after the class is created, to perform additional initialization before the class is handed to the caller:

```
# Metaprogramming/NewVSInit.py
from pprint import pprint

class Tag1: pass
class Tag2: pass
class Tag3:
    def tag3_method(self): pass

class MetaBase(type):
    def __new__(mcl, name, bases, nmspc):
        print('MetaBase.__new__\n')
        return super(MetaBase, mcl).__new__(mcl, name, bases, nmspc)

    def __init__(cls, name, bases, nmspc):
        print('MetaBase.__init__\n')
        super(MetaBase, cls).__init__(name, bases, nmspc)

class MetaNewVSInit(MetaBase):
    def __new__(mcl, name, bases, nmspc):
```

```
        # First argument is the metaclass ``MetaNewVSInit``
        print('MetaNewVSInit.__new__')
        for x in (mcl, name, bases, nmspc): pprint(x)
        print('')
        # These all work because the class hasn't been created yet:
        if 'foo' in nmspc: nmspc.pop('foo')
        name += '_x'
        bases += (Tag1,)
        nmspc['baz'] = 42
        return super(MetaNewVSInit, mcl).__new__(mcl, name, bases, nmspc)

    def __init__(cls, name, bases, nmspc):
        # First argument is the class being initialized
        print('MetaNewVSInit.__init__')
        for x in (cls, name, bases, nmspc): pprint(x)
        print('')
        if 'bar' in nmspc: nmspc.pop('bar') # No effect
        name += '_y' # No effect
        bases += (Tag2,) # No effect
        nmspc['pi'] = 3.14159 # No effect
        super(MetaNewVSInit, cls).__init__(name, bases, nmspc)
        # These do work because they operate on the class object:
        cls.__name__ += '_z'
        cls.__bases__ += (Tag3,)
        cls.e = 2.718

class Test(object):
    __metaclass__ = MetaNewVSInit
    def __init__(self):
        print('Test.__init__')
    def foo(self): print('foo still here')
    def bar(self): print('bar still here')

t = Test()
print('class name: ' + Test.__name__)
print('base classes: ', [c.__name__ for c in Test.__bases__])
print([m for m in dir(t) if not m.startswith("__")])
t.bar()
print(t.e)

""" Output:
MetaNewVSInit.__new__
<class '__main__.MetaNewVSInit'>
'Test'
(<type 'object'>,)
{'__init__': <function __init__ at 0x7ecf0>,
 '__metaclass__': <class '__main__.MetaNewVSInit'>,
 '__module__': '__main__',
 'bar': <function bar at 0x7ed70>,
 'foo': <function foo at 0x7ed30>}

MetaBase.__new__

MetaNewVSInit.__init__
<class '__main__.Test_x'>
'Test'
(<type 'object'>,)
{'__init__': <function __init__ at 0x7ecf0>,
```

```
 '__metaclass__': <class '__main__.MetaNewVSInit'>,
 '__module__': '__main__',
 'bar': <function bar at 0x7ed70>,
 'baz': 42}

MetaBase.__init__

Test.__init__
class name: Test_x_z
('base classes: ', ['object', 'Tag1', 'Tag3'])
['bar', 'baz', 'e', 'tag3_method']
bar still here
2.718
"""
```

The primary difference is that when overriding `__new__()` you can change things like the 'name', 'bases' and 'namespace' arguments before you call the super constructor and it will have an effect, but doing the same thing in `__init__()` you won't get any results from the constructor call.

One special case in `__new__()` is that you can manipulate things like `__slots__`, but in `__init__()` you can't.

Note that, since the base-class version of `__init__()` doesn't make any modifications, it makes sense to call it first, then perform any additional operations. In C++ and Java, the base-class constructor *must* be called as the first operation in a derived-class constructor, which makes sense because derived-class constructions can then build upon base-class foundations.

In many cases, the choice of `__new__()` vs `__init__()` is a style issue and doesn't matter, but because `__new__()` can do everything and `__init__()` is slightly more limited, some people just start using `__new__()` and stick with it. This use can be confusing – I tend to hunt for the reason that `__init__()` has been chosen, and if I can't find it wonder whether the author knew what they were doing. I prefer to only use `__new__()` when it has meaning – when you must in order to change things that only `__new__()` can change.

## Class Methods and Metamethods

A metamethod can be called from either the metaclass or from the class, but not from an instance. A classmethod can be called from either a class or its instances, but is not part of the metaclass.

(Is a similar relationship true with attributes, or is it different?)

### Intercepting Class Creation

This example implements *Singleton* using metaclasses, by overriding the `__call__()` metamethod, which is invoked when a new instance is created:

```
# Metaprogramming/Singleton.py

class Singleton(type):
    instance = None
    def __call__(cls, *args, **kw):
        if not cls.instance:
            cls.instance = super(Singleton, cls).__call__(*args, **kw)
        return cls.instance
```

```
class ASingleton(object):
    __metaclass__ = Singleton

a = ASingleton()
b = ASingleton()
assert a is b
print(a.__class__.__name__, b.__class__.__name__)

class BSingleton(object):
    __metaclass__ = Singleton

c = BSingleton()
d = BSingleton()
assert c is d
print(c.__class__.__name__, d.__class__.__name__)
assert c is not a

""" Output:
('ASingleton', 'ASingleton')
('BSingleton', 'BSingleton')
"""
```

By overriding `__call__()` in the metaclass, the creation of instances are intercepted. Instance creation is bypassed if one already exists.

Note the dependence upon the behavior of static class fields. When `cls.instance` is first read, it gets the static value of `instance` from the metaclass, which is `None`. However, when the assignment is made, Python creates a local version for the particular class, and the next time `cls.instance` is read, it sees that local version. Because of this behavior, each class ends up with its own class-specific `instance` field (thus `instance` is not somehow being "inherited" from the metaclass).

## A Class Decorator Singleton

```
# Metaprogramming/SingletonDecorator.py

def singleton(klass):
    "Simple replacement of object creation operation"
    def getinstance(*args, **kw):
        if not hasattr(klass, 'instance'):
            klass.instance = klass(*args, **kw)
        return klass.instance
    return getinstance

def singleton(klass):
    """
    More powerful approach: Change the behavior
    of the instances AND the class object.
    """
    class Decorated(klass):
        def __init__(self, *args, **kwargs):
            if hasattr(klass, '__init__'):
                klass.__init__(self, *args, **kwargs)
        def __repr__(self) : return klass.__name__ + " obj"
        __str__ = __repr__
    Decorated.__name__ = klass.__name__
    class ClassObject:
```

```
        def __init__(cls):
            cls.instance = None
        def __repr__(cls):
            return klass.__name__
        __str__ = __repr__
        def __call__(cls, *args, **kwargs):
            print str(cls) + " __call__ "
            if not cls.instance:
                cls.instance = Decorated(*args, **kwargs)
            return cls.instance
    return ClassObject()

@singleton
class ASingleton: pass

a = ASingleton()
b = ASingleton()
print(a, b)
print a.__class__.__name__
print ASingleton
assert a is b

@singleton
class BSingleton:
    def __init__(self, x):
        self.x = x

c = BSingleton(11)
d = BSingleton(22)
assert c is d
assert c is not a

""" Output:
ASingleton __call__
ASingleton __call__
(ASingleton obj, ASingleton obj)
ASingleton
ASingleton
BSingleton __call__
BSingleton __call__
"""
```

## The `__prepare__()` Metamethod

One of the things you *can't* do with class decorators is to replace the default dictionary. In Python 3 this is enabled with the `__prepare__()` metamethod:

```
@classmethod
def __prepare__(mcl, name, bases):
    return odict()
```

For an example of using both `__prepare__()` and `__slots__` in metaclasses, see Michele Simionato's article.

## Module-level `__metaclass__` Assignment

(Does this work in Python 3? If not is there an alternative?)

## Metaclass Conflicts

Note that the `metaclass` argument is singular – you can't attach more than one metaclass to a class. However, through multiple inheritance you can *accidentally* end up with more than one metaclass, and this produces a conflict which must be resolved.

http://code.activestate.com/recipes/204197/

## Further Reading

**Excellent step-by-step introduction to metaclasses:** http://cleverdevil.org/computing/78/

**Metaclass intro and comparison of syntax between Python 2.x and 3.x:** http://mikewatkins.ca/2008/11/29/python-2-and-3-metaclasses/

**David Mertz's metaclass primer:** http://www.onlamp.com/pub/a/python/2003/04/17/metaclasses.html

**Three-part in-depth coverage of metaclasses on IBM Developer Works. Quite useful and authoritative:**

- http://www.ibm.com/developerworks/linux/library/l-pymeta.html
- http://www.ibm.com/developerworks/linux/library/l-pymeta2/
- http://www.ibm.com/developerworks/linux/library/l-pymeta3.html

**Michele Simionato's articles on Artima, with special emphasis on the difference between Python 2.x and 3.x meta**

- http://www.artima.com/weblogs/viewpost.jsp?thread=236234
- http://www.artima.com/weblogs/viewpost.jsp?thread=236260

Once you understand the foundations, you can find lots of examples by searching for "metaclass" within the Python Cookbook: http://code.activestate.com/recipes/langs/python/

The printed version of the Python Cookbook has far fewer examples than the online version, but the print version has been filtered and edited and so tends to be more authoritative.

**Ian Bicking writes about metaclasses:**

- http://blog.ianbicking.org/a-conservative-metaclass.html
- http://blog.ianbicking.org/metaclass-fun.html
- http://blog.ianbicking.org/A-Declarative-Syntax-Extension.html
- http://blog.ianbicking.org/self-take-two.html

Lots of good information about classes, types, metaclasses, etc., including historical stuff in the Python 2.2 docs (is this duplicated in later versions of the docs):

- http://www.python.org/download/releases/2.2/descrintro/

For more advanced study, the book Putting Metaclasses to Work.

CHAPTER 16

# Generators, Iterators, and Itertools

CHAPTER 17

# Comprehensions

History: where did they come from?

They require a mind shift.

What makes them so compelling (once you 'get it')?

Comprehensions are constructs that allow sequences to be built from other sequences. Python 2.0 introduced list comprehensions and Python 3.0 comes with dictionary and set comprehensions.

## List Comprehensions

A list comprehension consists of the following parts:

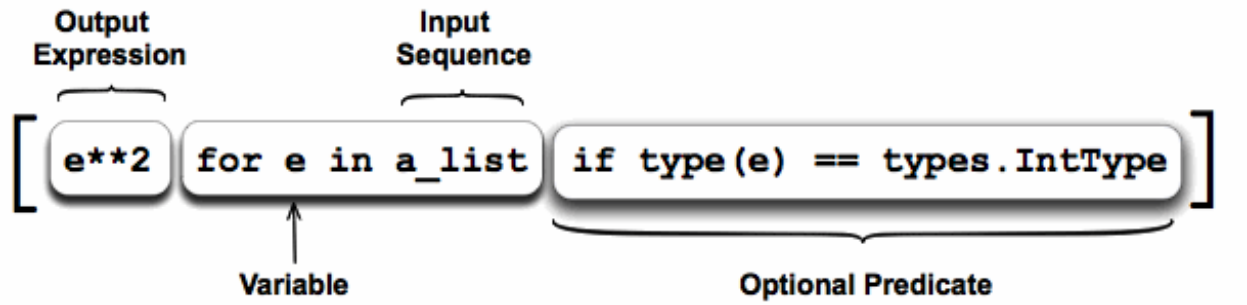
- An Input Sequence.
- A Variable representing members of the input sequence.
- An Optional Predicate expression.
- An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate.

Say we need to obtain a list of all the integers in a sequence and then square them:

```
a_list = [1, '4', 9, 'a', 0, 4]

squared_ints = [ e**2 for e in a_list if type(e) == types.IntType ]

print squared_ints
# [ 1, 81, 0, 16 ]
```




- The iterator part iterates through each member **e** of the input sequence **a_list**.
- The predicate checks if the member is an integer.
- If the member is an integer then it is passed to the output expression, squared, to become a member of the output list.

Much the same results can be achieved using the built in functions, **map**, **filter** and the anonymous **lambda** function.

The filter function applies a predicate to a sequence:

```
filter(lambda e: type(e) == types.IntType, a_list)
```

Map modifies each member of a sequence:

```
map(lambda e: e**2, a_list)
```

The two can be combined:

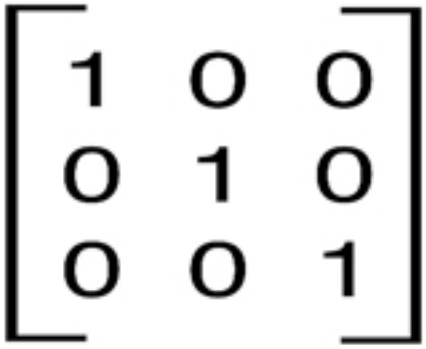
```
map(lambda e: e**2, filter(lambda e: type(e) == types.IntType, a_list))
```

The above example involves function calls to **map**, **filter**, **type** and two calls to **lambda**. Function calls in Python are expensive. Furthermore the input sequence is traversed through twice and an intermediate list is produced by filter.

The list comprehension is enclosed within a list so, it is immediately evident that a list is being produced. There is only one function call to **type** and no call to the cryptic **lambda** instead the list comprehension uses a conventional iterator, an expression and an if expression for the optional predicate.

## Nested Comprehensions

An identity matrix of size n is an n by n square matrix with ones on the main diagonal and zeros elsewhere. A 3 by 3 identity matrix is:



In python we can represent such a matrix by a list of lists, where each sub-list represents a row. A 3 by 3 matrix would be represented by the following list:

```
[ [ 1, 0, 0 ],
  [ 0, 1, 0 ],
  [ 0, 0, 1 ] ]
```

The above matrix can be generated by the following comprehension:

```
[ [ 1 if item_idx == row_idx else 0 for item_idx in range(0, 3) ] for row_idx in␣
↪range(0, 3) ]
```

## Techniques

Using `zip()` and dealing with two or more elements at a time:

```
['%s=%s' % (n, v) for n, v in zip(self.all_names, self)]
```

Multiple types (auto unpacking of a tuple):

```
[f(v) for (n, f), v in zip(cls.all_slots, values)]
```

A two-level list comprehension using `os.walk()`:

```
# Comprehensions/os_walk_comprehension.py
import os
restFiles = [os.path.join(d[0], f) for d in os.walk(".")
             for f in d[2] if f.endswith(".rst")]
for r in restFiles:
    print(r)
```

## A More Complex Example

**Note:** This will get a full description of all parts.

```
# CodeManager.py
"""
TODO: Break check into two pieces?
TODO: update() is still only in test mode; doesn't actually work yet.

Extracts, displays, checks and updates code examples in restructured text (.rst)
files.

You can just put in the codeMarker and the (indented) first line (containing the
file path) into your restructured text file, then run the update program to
automatically insert the rest of the file.
"""
import os, re, sys, shutil, inspect, difflib

restFiles = [os.path.join(d[0], f) for d in os.walk(".") if not "_test" in d[0]
             for f in d[2] if f.endswith(".rst")]

class Languages:
    "Strategy design pattern"

    class Python:
        codeMarker = "::\n\n"
        commentTag = "#"
        listings = re.compile("::\n\n( {4}#.*(?:\n+ {4}.*)*)")

    class Java:
        codeMarker = "..  code-block:: java\n\n"
        commentTag = "//"
        listings = \
            re.compile(".. *code-block:: *java\n\n( {4}//.*(?:\n+ {4}.*)*)")

def shift(listing):
    "Shift the listing left by 4 spaces"
    return [x[4:] if x.startswith("    ") else x for x in listing.splitlines()]

# TEST - makes duplicates of the rst files in a test directory to test update():
dirs = set([os.path.join("_test", os.path.dirname(f)) for f in restFiles])
if [os.makedirs(d) for d in dirs if not os.path.exists(d)]:
    [shutil.copy(f, os.path.join("_test", f)) for f in restFiles]
testFiles = [os.path.join(d[0], f) for d in os.walk("_test")
             for f in d[2] if f.endswith(".rst")]

class Commands:
    """
    Each static method can be called from the command line. Add a new static
    method here to add a new command to the program.
    """

    @staticmethod
    def display(language):
        """
        Print all the code listings in the .rst files.
```

```
        """
        for f in restFiles:
            listings = language.listings.findall(open(f).read())
            if not listings: continue
            print('=' * 60 + "\n" + f + "\n" + '=' * 60)
            for n, l in enumerate(listings):
                print("\n".join(shift(l)))
                if n < len(listings) - 1:
                    print('-' * 60)

    @staticmethod
    def extract(language):
        """
        Pull the code listings from the .rst files and write each listing into
        its own file. Will not overwrite if code files and .rst files disagree
        unless you say "extract -force".
        """
        force = len(sys.argv) == 3 and sys.argv[2] == '-force'
        paths = set()
        for listing in [shift(listing) for f in restFiles
                        for listing in language.listings.findall(open(f).read())]:
            path = listing[0][len(language.commentTag):].strip()
            if path in paths:
                print("ERROR: Duplicate file name: %s" % path)
                sys.exit(1)
            else:
                paths.add(path)
            path = os.path.join("..", "code", path)
            dirname = os.path.dirname(path)
            if dirname and not os.path.exists(dirname):
                os.makedirs(dirname)
            if os.path.exists(path) and not force:
                for i in difflib.ndiff(open(path).read().splitlines(), listing):
                    if i.startswith("+ ") or i.startswith("- "):
                        print("ERROR: Existing file different from .rst")
                        print("Use 'extract -force' to force overwrite")
                        Commands.check(language)
                        return
            file(path, 'w').write("\n".join(listing))

    @staticmethod
    def check(language):
        """
        Ensure that external code files exist and check which external files
        have changed from what's in the .rst files. Generate files in the
        _deltas subdirectory showing what has changed.
        """
        class Result: # Messenger
            def __init__(self, **kwargs):
                self.__dict__ = kwargs
        result = Result(missing = [], deltas = [])
        listings = [Result(code = shift(code), file = f)
                    for f in restFiles for code in
                    language.listings.findall(open(f).read())]
        paths = [os.path.normpath(os.path.join("..", "code", path)) for path in
                    [listing.code[0].strip()[len(language.commentTag):].strip()
                     for listing in listings]]
        if os.path.exists("_deltas"):
```

```
            shutil.rmtree("_deltas")
        for path, listing in zip(paths, listings):
            if not os.path.exists(path):
                result.missing.append(path)
            else:
                code = open(path).read().splitlines()
                for i in difflib.ndiff(listing.code, code):
                    if i.startswith("+ ") or i.startswith("- "):
                        d = difflib.HtmlDiff()
                        if not os.path.exists("_deltas"):
                            os.makedirs("_deltas")
                        html = os.path.join("_deltas",
                            os.path.basename(path).split('.')[0] + ".html")
                        open(html, 'w').write(
                            "<html><h1>Left: %s<br>Right: %s</h1>" %
                            (listing.file, path) +
                            d.make_file(listing.code, code))
                        result.deltas.append(Result(file = listing.file,
                            path = path, html = html, code = code))
                        break
        if result.missing:
            print("Missing %s files:\n%s" %
                  (language.__name__, "\n".join(result.missing)))
        for delta in result.deltas:
            print("%s changed in %s; see %s" %
                  (delta.file, delta.path, delta.html))
        return result

    @staticmethod
    def update(language): # Test until it is trustworthy
        """
        Refresh external code files into .rst files.
        """
        check_result = Commands.check(language)
        if check_result.missing:
            print(language.__name__, "update aborted")
            return
        changed = False
        def _update(matchobj):
            listing = shift(matchobj.group(1))
            path = listing[0].strip()[len(language.commentTag):].strip()
            filename = os.path.basename(path).split('.')[0]
            path = os.path.join("..", "code", path)
            code = open(path).read().splitlines()
            return language.codeMarker + \
                "\n".join([("    " + line).rstrip() for line in listing])
        for f in testFiles:
            updated = language.listings.sub(_update, open(f).read())
            open(f, 'w').write(updated)

if __name__ == "__main__":
    commands = dict(inspect.getmembers(Commands, inspect.isfunction))
    if len(sys.argv) < 2 or sys.argv[1] not in commands:
        print("Command line options:\n")
        for name in commands:
            print(name + ": " + commands[name].__doc__)
    else:
        for language in inspect.getmembers(Languages, inspect.isclass):
```

```
            commands[sys.argv[1]](language[1])
```

## Set Comprehensions

Set comprehensions allow sets to be constructed using the same principles as list comprehensions, the only difference is that resulting sequence is a set.

Say we have a list of names. The list can contain names which only differ in the case used to represent them, duplicates and names consisting of only one character. We are only interested in names longer then one character and wish to represent all names in the same format: The first letter should be capitalised, all other characters should be lower case.

Given the list:

```
names = [ 'Bob', 'JOHN', 'alice', 'bob', 'ALICE', 'J', 'Bob' ]
```

We require the set:

```
{ 'Bob', 'John', 'Alice' }
```

Note the new syntax for denoting a set. Members are enclosed in curly braces.

The following set comprehension accomplishes this:

```
{ name[0].upper() + name[1:].lower() for name in names if len(name) > 1 }
```

## Dictionary Comprehensions

Say we have a dictionary the keys of which are characters and the values of which map to the number of times that character appears in some text. The dictionary currently distinguishes between upper and lower case characters.

We require a dictionary in which the occurrences of upper and lower case characters are combined:

```
mcase = {'a':10, 'b': 34, 'A': 7, 'Z':3}

mcase_frequency = { k.lower() : mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0) for␣
↪k in mcase.keys() }

# mcase_frequency == {'a': 17, 'z': 3, 'b': 34}
```

**Note:** Contributions by Michael Charlton, 3/23/09

# CHAPTER 18

# Coroutines, Concurrency & Distributed Systems

[[ Will probably need to expand this to multiple chapters:

1. Concurrency Concepts
2. Coroutines
3. Processes
4. Threads

(this isn't final; may need different organization or finer grained. However, it should start with simpler concepts and progress to the more difficult ones, as above.)

]]

Primary focus should be on:

1. Using `yield` to create coroutines
2. Using the new `multiprocessing` module

and then showing some alternative techniques.

foo bar `input()` baz.

## The GIL

The GIL prevents context switches from happening in the middle of C code. Basically, it makes any C code into a critical section, except when that C code explicitly releases the GIL. This greatly simplifies the task of writing extension modules as well the Python core.

The designers of Python made a design decision that extension writers would not have to take care of locking. Thus, Python is intended to be simple/easy to integrate with any C library. In order to remove the GIL, you'd have to go into all existing C code and write explicit locking/unlocking code, and you'd have to do this with every new C library as well.

[[ Description of how it supports/impacts reference-counted garbage collection]]

## Multiprocessing

Example by Michele Simionato in comp lang python. Here is an example of using multiprocessing (which is included in Python 2.6 and easy_installable in older Python versions) to print a spin bar while a computation is running:

```
import sys, time
import multiprocessing
DELAY = 0.1
DISPLAY = [ '|', '/', '-', '\\' ]
def spinner_func(before='', after=''):
    write, flush = sys.stdout.write, sys.stdout.flush
    pos = -1
    while True:
        pos = (pos + 1) % len(DISPLAY)
        msg = before + DISPLAY[pos] + after
        write(msg); flush()
        write('\x08' * len(msg))
        time.sleep(DELAY)
def long_computation():
    # emulate a long computation
    time.sleep(3)
if __name__ == '__main__':
    spinner = multiprocessing.Process(
        None, spinner_func, args=('Please wait ... ', ''))
    spinner.start()
    try:
        long_computation()
        print 'Computation done'
    finally:
        spinner.terminate()
```

On the Erlang mail list, four years ago, Erlang expert Joe Armstrong posted this:

> In Concurrency Oriented (CO) programming you concentrate on the concurrency and the messages between the processes. There is no sharing of data.
>
> [A program] should be thought of thousands of little black boxes all doing things in parallel - these black boxes can send and receive messages. Black boxes can detect errors in other black boxes - that's all. ... Erlang uses a simple functional language inside the [black boxes] - this is not particularly interesting - *any* language that does the job would do - the important bit is the concurrency.

On the Squeak mail list in 1998, Alan Kay had this to say:

> ...Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.
>
> The big idea is "messaging" – that is what the kernal of Smalltalk/Squeak is all about... The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be. Think of the internet – to live, it (a) has to allow many different kinds of ideas and realizations that are beyond any single standard and (b) to allow varying degrees of safe interoperability between these ideas.
>
> If you focus on just messaging – and realize that a good metasystem can late bind the various 2nd level architectures used in objects – then much of the language-, UI-, and OS based discussions on this thread are really quite moot.

## Further Reading

This article argues that large-scale parallelism – which is what `multiprocessing` supports – is the more important problem to solve, and that functional languages don't help that much with this problem.

http://jessenoller.com/2009/02/01/python-threads-and-the-global-interpreter-lock/

# CHAPTER 19

# Jython

**Note:** This chapter is being brought up to date with Jython 2.5, and will need changes when Jython 3 comes out.

**Note:** Some of the descriptions in this chapter are introductory, so that the material can be used to introduce Java programmers to Jython.

Sometimes it's easier and faster to temporarily step into another language to solve a particular aspect of your problem.

This chapter looks at the value of crossing language boundaries. It is often advantageous to solve a problem using more than one programming language; as you'll see, a problem that is very difficult or tedious to solve in one language can often be solved quickly and easily in another. By combining languages, you can create your product much more quickly and cheaply.

One use of this idea is the *Interpreter* design pattern, which adds an interpreted language to your program to allow the end user to easily customize a solution. If the application user needs greater run time flexibility, for example to create scripts describing the desired behavior of the system, you can use *Interpreter* by creating and embedding a language interpreter into your program.

In Java, the easiest and most powerful way to do this is with *Jython*[1], an implementation of Python in pure Java byte codes. As you will see, this brings together the benefits of both worlds.

Jython is generated entirely in Java byte codes, so incorporating it into your application is quite simple, and it's as portable as Java is. It has an extremely clean interface with Java: Java can call Python classes, and Python can call Java classes.

Because Jython is just Java classes, it can often be "stealthed" into companies that have rigid processes for using new languges and tools. If Java has been accepted, such companies often accept anything that runs on the JVM without question.

[1] The original version of this was called *JPython*, but the project changed and the name was changed to emphasize the distinctness of the new version.

The Python/Jython language can be freely embedded into your for-profit application without signing any license agreement, paying royalties, or dealing with strings of any kind. There are basically no restrictions when you're using Python/Jython.

Python is designed with classes from the ground up and provides pure support for object-oriented programming (both C++ and Java violate purity in various ways). Python scales up so that you can create large programs without losing control of the code. Java projects have been quickly created using Jython, then later optimized by rewriting into Java sections of the Jython code that have profiled as bottlenecks.

## Installation

To install Jython, go to http://jython.sourceforge.net.

**Note:** Select "test the beta".

The download is a **.class** file, which will run an installer when you execute it using `java -jar`.

You also need the Java Development Kit (JDK), and to add **jython-complete.jar** to your Java CLASSPATH. As an example, here is the appropriate section in my `.bashrc` for ***nix**; for Windows you need to do the equivalent:

```
export set JYTHON_HOME="/Users/bruceeckel/jython2.5b0"
export set CLASSPATH=.:.:.:$JYTHON_HOME/jython-complete.jar
```

When you run Jython, you might get the warning: `can't create package cache dir, '/cachedir/packages'`. Jython will still work, but startup will be slower because caching isn't happening. Jython caching requires `/cachedir/packages/` in the `python.home` directory. It is often the case on ***nix** that users lack sufficient priveleges to create or write to this directory. Because the problem is merely permissions, something like `mkdir cachedir; chmod a+rw cachedir` within the Jython directory should eliminate this warning message.

### Getting the Trunk

**Note:** This section has not been successfuly tested yet.

The Jython development trunk is very stable so it's safe to get as the most recent version of the implementation. The subversion command is:

```
svn co https://jython.svn.sourceforge.net/svnroot/jython/trunk/jython
```

Then just invoke `ant` against the `build.xml` file. `dist/bin/jython` is a shell script that starts up jython in console mode. Lastly, modify the registry (in dist/registry) so that:

```
python.console=org.python.util.ReadlineConsole
python.console.readlinelib=GnuReadline
```

(`readline` is GPL, so it makes it a bit harder to automate this part of the distro). See: http://wiki.python.org/jython/ReadlineSetup

## Scripting

One compelling benefit of using a dynamic language on the JVM is scripting. You can rapidly create and test code, and solve problems more quickly.

Here's an example that shows a little of what you can do in a Jython script, and also gives you a sense of performance:

```
# Jython/Simple.py
import platform, glob, time
from subprocess import Popen, PIPE

print platform.uname() # What are we running on?
print glob.glob("*.py") # Find files with .py extensions
# Send a command to the OS and capture the results:
print Popen(["ping", "-c", "1", "www.mindview.net"],
               stdout=PIPE).communicate()[0]
# Time an operation:
start = time.time()
for n in xrange(1000000):
    for i in xrange(10):
            oct(i)
print time.time() - start
```

**Note:** The `timeit` module in the alpha distribution could not be used as it tries to turn off the Java garbage collector.

If you run this program under both cpython and Jython, you'll see that the timed loop produces very similar results; Jython 2.5 is in beta so this is quite impressive and should get faster – there's even talk that Jython could run faster than cpython, because of the optimization benefits of the JVM. The total runtime of the cpython version is faster because of its rapid startup time; the JVM always has a delay for startup.

Note that things that are very quick to write in Jython require much more code (and often research) in Java. Here's an example that uses a Python *list comprehension* with the **os.walk()** function to visit all the directories in a directory tree, and find all the files with names that end in **.java** and contain the word **PythonInterpreter**:

```
# Jython/Walk_comprehension.py
import os

restFiles = [os.path.join(d[0], f) for d in os.walk(".")
             for f in d[2] if f.endswith(".java") and
             "PythonInterpreter" in open(os.path.join(d[0], f)).read()]

for r in restFiles:
    print(r)
```

You can certainly achieve this in Java. It will just take a lot longer.

Often more sophisticated programs begin as scripts, and then evolve. The fact that you can quickly try things out allows you to test concepts, and then create more refined code as needed.

# Interpreter Motivation

Remember that each design pattern allows one or more factors to change, so it's important to first be aware of which factor is changing. Sometimes the end users of your application (rather than the programmers of that application) need complete flexibility in the way that they configure some aspect of the program. That is, they need to do some kind of simple programming. The *Interpreter* pattern provides this flexibility by adding a language interpreter.

The problem is that creating your own language and building an interpreter is a time-consuming distraction from the process of developing your application. You must ask whether you want to finish writing your application or make a new language. The best solution is to reuse code: embed an interpreter that's already been built and debugged for you.

## Creating a Language

It turns out to be remarkably simple to use Jython to create an interpreted language inside your application. Consider the greenhouse controller example from *Thinking in Java*. This is a situation where you want the end user – the person managing the greenhouse – to have configuration control over the system, and so a simple scripting language is an ideal solution. This is often called a *domain-specific language* (DSL) because it solves a particular domain problem.

To create the language, we'll simply write a set of Python classes, and the constructor of each will add itself to a (static) master list. The common data and behavior will be factored into the base class **Event**. Each **Event** object will contain an **action** string (for simplicity – in reality, you'd have some sort of functionality) and a time when the event is supposed to run. The constructor initializes these fields, and then adds the new **Event** object to a static list called **events** (defining it in the class, but outside of any methods, is what makes it static):

```
# Jython/GreenHouseLanguage.py

class Event:
    events = [] # static

    def __init__(self, action, time):
        self.action = action
        self.time = time
        Event.events.append(self)

    def __cmp__ (self, other):
        "So sort() will compare only on time."
        return cmp(self.time, other.time)

    def run(self):
        print("%.2f: %s" % (self.time, self.action))

    @staticmethod
    def run_events():
        Event.events.sort();
        for e in Event.events:
            e.run()

class LightOn(Event):
    def __init__(self, time):
        Event.__init__(self, "Light on", time)

class LightOff(Event):
```

```
    def __init__(self, time):
        Event.__init__(self, "Light off", time)

class WaterOn(Event):
    def __init__(self, time):
        Event.__init__(self, "Water on", time)

class WaterOff(Event):
    def __init__(self, time):
        Event.__init__(self, "Water off", time)

class ThermostatNight(Event):
    def __init__(self, time):
        Event.__init__(self,"Thermostat night", time)

class ThermostatDay(Event):
    def __init__(self, time):
        Event.__init__(self, "Thermostat day", time)

class Bell(Event):
    def __init__(self, time):
        Event.__init__(self, "Ring bell", time)

if __name__ == "__main__":
    ThermostatNight(5.00)
    LightOff(2.00)
    WaterOn(3.30)
    WaterOff(4.45)
    LightOn(1.00)
    ThermostatDay(6.00)
    Bell(7.00)
    Event.run_events()
```

**Note:** To run this program say `python GreenHouseLanguage.py` or `jython GreenHouseLanguage.py`.

The constructor of each derived class calls the base-class constructor, which adds the new object to the list. The **run()** function sorts the list, which automatically uses the **__cmp__()** method defined in **Event** to base comparisons on time only. In this example, it only prints out the list, but in the real system it would wait for the time of each event to come up and then run the event.

The **__main__** section performs a simple test on the classes.

The above file – which is an ordinary Python program – is now a module that can be included in another Python program. But instead of using it in an ordinary Python program, let's use Jython, inside of Java. This turns out to be remarkably simple: you import some Jython classes, create a **PythonInterpreter** object, and cause the Python files to be loaded:

```
// Jython/GreenHouseController.java
import org.python.core.*;
import org.python.util.PythonInterpreter;

public class GreenHouseController {
  public static void main(String[] args) throws PyException  {
    PythonInterpreter interp = new PythonInterpreter();
    System.out.println("Loading GreenHouse Language");
```

```
    interp.execfile("GreenHouseLanguage.py");
    System.out.println("Loading GreenHouse Script");
    interp.execfile("Schedule.ghs");
    System.out.println("Executing GreenHouse Script");
    interp.exec("run()");
  }
}
```

The **PythonInterpreter** object is a complete Python interpreter that accepts commands from the Java program. One of these commands is **execfile()**, which tells it to execute all the statements it finds in a particular file. By executing **GreenHouseLanguage.py**, all the classes from that file are loaded into our **PythonInterpreter** object, and so it now "holds" the greenhouse controller language. The **Schedule.ghs** file is the one created by the end user to control the greenhouse. Here's an example:

```
# Jython/Schedule.ghs
Bell(7.00)
ThermostatDay(6.00)
WaterOn(3.30)
LightOn(1.00)
ThermostatNight(5.00)
LightOff(2.00)
WaterOff(4.45)
```

This is the goal of the interpreter design pattern: to make the configuration of your program as simple as possible for the end user. With Jython you can achieve this with almost no effort at all.

One of the other methods available to the **PythonInterpreter** is **exec()**, which allows you to send a command to the interpreter. In the above program, the **run()** function is called using **exec()**.

## Using Java libraries

Jython wraps Java libraries so that any of them can be used directly or via inheritance. In addition, Python shorthand simplifies coding.

As an example, consider the **HTMLButton.java** example from *Thinking in Java*. Here is its conversion to Jython:

```
# Jython/PythonSwing.py
# The HTMLButton.java example from "Thinking in Java"
# converted into Jython.
from javax.swing import JFrame, JButton, JLabel
from java.awt import FlowLayout

frame = JFrame("HTMLButton", visible=1,
  defaultCloseOperation=JFrame.EXIT_ON_CLOSE)

def kapow(e):
    frame.contentPane.add(JLabel("<html>"+
      "<i><font size=+4>Kapow!"))
    # Force a re-layout to
    # include the new label:
    frame.validate()

button = JButton("<html><b><font size=+2>" +
  "<center>Hello!<br><i>Press me now!",
  actionPerformed=kapow)
```

```
frame.contentPane.layout = FlowLayout()
frame.contentPane.add(button)
frame.pack()
frame.size=200, 500
```

If you compare the Java version of the program to the above Jython implementation, you'll see that Jython is shorter and generally easier to understand. For example, to set up the frame in the Java version you had to make several calls: the constructor for **JFrame()**, the **setVisible()** method and the **setDefaultCloseOperation()** method, whereas in the above code all three of these operations are performed with a single constructor call.

Also notice that the **JButton** is configured with an **actionListener()** method inside the constructor, with the assignment to **kapow**. In addition, Jython's JavaBean awareness means that a call to any method with a name that begins with "**set**" can be replaced with an assignment, as you see above.

The only method that did not come over from Java is the **pack()** method, which seems to be essential in order to force the layout to happen properly. It's also important that the call to **pack()** appear *before* the **size** setting.

## Inheriting from Java library Classes

You can easily inherit from standard Java library classes in Jython. Here's the **Dialogs.java** example from *Thinking in Java*, converted into Jython:

```
# Jython/PythonDialogs.py
# Dialogs.java from "Thinking in Java" converted into Jython.
from java.awt import FlowLayout
from javax.swing import JFrame, JDialog, JLabel
from javax.swing import JButton

class MyDialog(JDialog):
    def __init__(self, parent=None):
        JDialog.__init__(self, title="My dialog", modal=1)
        self.contentPane.layout = FlowLayout()
        self.contentPane.add(JLabel("A dialog!"))
        self.contentPane.add(JButton("OK",
          actionPerformed =
            lambda e, t=self: t.dispose()))
        self.pack()

frame = JFrame("Dialogs", visible=1,
  defaultCloseOperation=JFrame.EXIT_ON_CLOSE)
dlg = MyDialog()
frame.contentPane.add(
  JButton("Press here to get a Dialog Box",
    actionPerformed = lambda e: dlg.show()))
frame.pack()
```

**MyDialog** is inherited from **JDialog**, and you can see named arguments being used in the call to the base-class constructor.

In the creation of the "OK" **JButton**, note that the **actionPerformed** method is set right inside the constructor, and that the function is created using the Python **lambda** keyword. This creates a nameless function with the arguments appearing before the colon and the expression that generates the returned value after the colon. As you should know, the Java prototype for the **actionPerformed()** method only contains a single argument, but the lambda expression indicates two. However, the second argument is provided with a

default value, so the function *can* be called with only one argument. The reason for the second argument is seen in the default value, because this is a way to pass **self** into the lambda expression, so that it can be used to dispose of the dialog.

Compare this code with the version that's published in *Thinking in Java*. You'll find that Python language features allow a much more succinct and direct implementation.

## Controlling Java from Jython

There's a tremendous amount that you can accomplish by controlling Python from Java. But one of the amazing things about Jython is that it makes Java classes almost transparently available from within Jython. Basically, a Java class looks like a Python class. This is true for standard Java library classes as well as classes that you create yourself, as you can see here:

```
# Jython/JavaClassInPython.py
# Using Java classes within Jython
# run with: jython.bat JavaClassInPython.py
from java.util import Date, HashSet, HashMap
from Jython.javaclass import JavaClass
from math import sin

d = Date() # Creating a Java Date object
print(d) # Calls toString()

# A "generator" to easily create data:
class ValGen:
    def __init__(self, maxVal):
        self.val = range(maxVal)
    # Called during 'for' iteration:
    def __getitem__(self, i):
        # Returns a tuple of two elements:
        return self.val[i], sin(self.val[i])

# Java standard containers:
jmap = HashMap()
jset = HashSet()

for x, y in ValGen(10):
    jmap.put(x, y)
    jset.add(y)
    jset.add(y)

print(jmap)
print(jset)

# Iterating through a set:
for z in jset:
    print(z, z.__class__)

print(jmap[3]) # Uses Python dictionary indexing
for x in jmap.keySet(): # keySet() is a Map method
    print(x, jmap[x])

# Using a Java class that you create yourself is
# just as easy:
jc = JavaClass()
```

```
jc2 = JavaClass("Created within Jython")
print(jc2.getVal())
jc.setVal("Using a Java class is trivial")
print(jc.getVal())
print(jc.getChars())
jc.val = "Using bean properties"
print(jc.val)
```

**Todo**

rewrite to distinguish python generator from above description, or choose different name.

Note that the **import** statements map to the Java package structure exactly as you would expect. In the first example, a **Date()** object is created as if it were a native Python class, and printing this object just calls **toString()**.

**ValGen** implements the concept of a "generator" which is used a great deal in the C++ STL (*Standard Template Library*, part of the Standard C++ Library). A generator is an object that produces a new object every time its "generation method" is called, and it is quite convenient for filling containers. Here, I wanted to use it in a **for** iteration, and so I needed the generation method to be the one that is called by the iteration process. This is a special method called **__getitem__()**, which is actually the overloaded operator for indexing, '[ ]'. A **for** loop calls this method every time it wants to move the iteration forward, and when the elements run out, **__getitem__()** throws an out-of-bounds exception and that signals the end of the **for** loop (in other languages, you would never use an exception for ordinary control flow, but in Python it seems to work quite well). This exception happens automatically when **self.val[i]** runs out of elements, so the **__getitem__()** code turns out to be simple. The only complexity is that **__getitem__()** appears to return *two* objects instead of just one. What Python does is automatically package multiple return values into a tuple, so you still only end up returning a single object (in C++ or Java you would have to create your own data structure to accomplish this). In addition, in the **for** loop where **ValGen** is used, Python automatically "unpacks" the tuple so that you can have multiple iterators in the **for**. These are the kinds of syntax simplifications that make Python so endearing.

The **jmap** and **jset** objects are instances of Java's **HashMap** and **HashSet**, again created as if those classes were just native Python components. In the **for** loop, the **put()** and **add()** methods work just like they do in Java. Also, indexing into a Java **Map** uses the same notation as for dictionaries, but note that to iterate through the keys in a **Map** you must use the **Map** method **keySet()** rather than the Python dictionary method **keys()**.

The final part of the example shows the use of a Java class that I created from scratch, to demonstrate how trivial it is. Notice also that Jython intuitively understands JavaBeans properties, since you can either use the **getVal()** and **setVal()** methods, or assign to and read from the equivalent **val** property. Also, **getChars()** returns a **Character[]** in Java, and this automatically becomes an array in Python.

The easiest way to use Java classes that you create for use inside a Python program is to put them inside a package. Although Jython can also import unpackaged java classes (**import JavaClass**), all such unpackaged java classes will be treated as if they were defined in different packages so they can only see each other's public methods.

Java packages translate into Jython modules, and Jython must import a module in order to be able to use the Java class. Here is the Java code for **JavaClass**:

```
// Jython/javaclass/JavaClass.java
package Jython.javaclass;
import java.util.*;

public class JavaClass {
```

```
  private String s = "";
  public JavaClass() {
    System.out.println("JavaClass()");
  }
  public JavaClass(String a) {
    s = a;
    System.out.println("JavaClass(String)");
  }
  public String getVal() {
    System.out.println("getVal()");
    return s;
  }
  public void setVal(String a) {
    System.out.println("setVal()");
    s = a;
  }
  public Character[] getChars() {
    System.out.println("getChars()");
    Character[] r = new Character[s.length()];
    for(int i = 0; i < s.length(); i++)
      r[i] = new Character(s.charAt(i));
    return r;
  }
  public static void main(String[] args) {
    JavaClass
      x1 = new JavaClass(),
      x2 = new JavaClass("UnitTest");
    System.out.println(x2.getVal());
    x1.setVal("SpamEggsSausageAndSpam");
    System.out.println(Arrays.toString(x1.getChars()));
  }
}
```

You can see that this is just an ordinary Java class, without any awareness that it will be used in a Jython program. For this reason, one of the important uses of Jython is in testing Java code[2]. Because Python is such a powerful, flexible, dynamic language it is an ideal tool for automated test frameworks, without making any changes to the Java code that's being tested.

## Inner Classes

Inner classes becomes attributes on the class object. Instances of **static** inner classes can be created with the usual call:

```
com.foo.JavaClass.StaticInnerClass()
```

Non-**static** inner classes must have an outer class instance supplied explicitly as the first argument:

```
com.foo.JavaClass.InnerClass(com.foo.JavaClass())
```

[2] Changing the registry setting **python.security.respectJavaAccessibility = true** to **false** makes testing even more powerful because it allows the test script to use *all* methods, even protected and package- private.

## Controlling the Interpreter

In the rest of this chapter, we shall look at more sophisticated ways to interact with Jython. The simplest way to exercise more control over the **PythonInterpreter** object from within Java is to send data to the interpreter, and pull data back out.

### Putting Data In

To inject data into your Python program, the **PythonInterpreter** class has a deceptively simple method: **set()**. However, **set()** takes many different data types and performs conversions upon them. The following example is a reasonably thorough exercise of the various **set()** possibilities, along with comments that should give a fairly complete explanation:

```
// Jython/PythonInterpreterSetting.java
// Passing data from Java to python when using
// the PythonInterpreter object.
import org.python.util.PythonInterpreter;
import org.python.core.*;
import java.util.*;

public class PythonInterpreterSetting {
  public static void main(String[] args) throws PyException  {
    PythonInterpreter interp = new PythonInterpreter();
    // It automatically converts Strings
    // into native Python strings:
    interp.set("a", "This is a test");
    interp.exec("print(a)");
    interp.exec("print(a[5:])"); // A slice
    // It also knows what to do with arrays:
    String[] s = { "How", "Do", "You", "Do?" };
    interp.set("b", s);
    interp.exec("for x in b: print(x[0], x)");
    // set() only takes Objects, so it can't
    // figure out primitives. Instead,
    // you have to use wrappers:
    interp.set("c", new PyInteger(1));
    interp.set("d", new PyFloat(2.2));
    interp.exec("print(c + d)");
    // You can also use Java's object wrappers:
    interp.set("c", new Integer(9));
    interp.set("d", new Float(3.14));
    interp.exec("print(c + d)");
    // Define a Python function to print arrays:
    interp.exec(
      "def prt(x): \n" +
      "  print(x)\n" +
      "  for i in x: \n" +
      "    print(i,)\n" +
      "  print(x.__class__)\n");
    // Arrays are Objects, so it has no trouble
    // figuring out the types contained in arrays:
    Object[] types = {
      new boolean[]{ true, false, false, true },
      new char[]{ 'a', 'b', 'c', 'd' },
      new byte[]{ 1, 2, 3, 4 },
      new int[]{ 10, 20, 30, 40 },
```

```
      new long[]{ 100, 200, 300, 400 },
      new float[]{ 1.1f, 2.2f, 3.3f, 4.4f },
      new double[]{ 1.1, 2.2, 3.3, 4.4 },
    };
    for(int i = 0; i < types.length; i++) {
      interp.set("e", types[i]);
      interp.exec("prt(e)");
    }
    // It uses toString() to print Java objects:
    interp.set("f", new Date());
    interp.exec("print(f)");
    // You can pass it a List
    // and index into it...
    List x = new ArrayList();
    for(int i = 0; i < 10; i++)
        x.add(new Integer(i * 10));
    interp.set("g", x);
    interp.exec("print(g)");
    interp.exec("print(g[1])");
    // ... But it's not quite smart enough
    // to treat it as a Python array:
    interp.exec("print(g.__class__)");
    // interp.exec("print(g[5:])"); // Fails
    // must extract the Java array:
    System.out.println("ArrayList to array:");
    interp.set("h", x.toArray());
    interp.exec("print(h.__class__)");
    interp.exec("print(h[5:])");
    // Passing in a Map:
    Map m = new HashMap();
    m.put(new Integer(1), new Character('a'));
    m.put(new Integer(3), new Character('b'));
    m.put(new Integer(5), new Character('c'));
    m.put(new Integer(7), new Character('d'));
    m.put(new Integer(11), new Character('e'));
    System.out.println("m: " + m);
    interp.set("m", m);
    interp.exec("print(m, m.__class__," +
      "m[1], m[1].__class__)");
    // Not a Python dictionary, so this fails:
    //! interp.exec("for x in m.keys():" +
    //!   "print(x, m[x])");
    // To convert a Map to a Python dictionary, use PyUtil:
    interp.set("m", PyUtil.toPyDictionary(m));
    interp.exec("print(m, m.__class__, " +
      "m[1], m[1].__class__)");
    interp.exec("for x in m.keys():print(x,m[x])");
  }
}
```

As usual with Java, the distinction between real objects and primitive types causes trouble. In general, if you pass a regular object to **set()**, it knows what to do with it, but if you want to pass in a primitive you must perform a conversion. One way to do this is to create a "Py" type, such as **PyInteger** or **PyFloat**. but it turns out you can also use Java's own object wrappers like **Integer** and **Float**, which is probably going to be a lot easier to remember.

Early in the program you'll see an **exec()** containing the Python statement:

```
print(a[5:])
```

The colon inside the indexing statement indicates a Python *slice*, which produces a range of elements from the original array. In this case, it produces an array containing the elements from number 5 until the end of the array. You could also say '**a[3:5]**' to produce elements 3 through 5, or '**a[:5]**' to produce the elements zero through 5. The reason a slice is used in this statement is to make sure that the Java **String** has really been converted to a Python string, which can also be treated as an array of characters.

You can see that it's possible, using **exec()**, to create a Python function (although it's a bit awkward). The **prt()** function prints the whole array, and then (to make sure it's a real Python array), iterates through each element of the array and prints it. Finally, it prints the class of the array, so we can see what conversion has taken place (Python not only has run-time type information, it also has the equivalent of Java reflection). The **prt()** function is used to print arrays that come from each of the Java primitive types.

Although a Java **ArrayList** does pass into the interpreter using **set()**, and you can index into it as if it were an array, trying to create a slice fails. To completely convert it into an array, one approach is to simply extract a Java array using **toArray()**, and pass that in. The **set()** method converts it to a **PyArray** – one of the classes provided with Jython – which can be treated as a Python array (you can also explicitly create a **PyArray**, but this seems unnecessary).

Finally, a **Map** is created and passed directly into the interpreter. While it is possible to do simple things like index into the resulting object, it's not a real Python dictionary so you can't (for example) call the **keys()** method. There is no straightforward way to convert a Java **Map** into a Python dictionary, and so I wrote a utility called **toPyDictionary()** and made it a **static** method of **net.mindview.python.PyUtil**. This also includes utilities to extract a Python array into a Java **List**, and a Python dictionary into a Java **Map**:

```
// Jython/PyUtil.java
// PythonInterpreter utilities
import org.python.util.PythonInterpreter;
import org.python.core.*;
import java.util.*;

public class PyUtil {
  /** Extract a Python tuple or array into a Java
  List (which can be converted into other kinds
  of lists and sets inside Java).
  @param interp The Python interpreter object
  @param pyName The id of the python list object
  */
  public static List
  toList(PythonInterpreter interp, String pyName){
    return new ArrayList(Arrays.asList(
      (Object[])interp.get(
        pyName, Object[].class)));
  }
  /** Extract a Python dictionary into a Java Map
  @param interp The Python interpreter object
  @param pyName The id of the python dictionary
  */
  public static Map
  toMap(PythonInterpreter interp, String pyName){
    PyList pa = ((PyDictionary)interp.get(
      pyName)).items();
    Map map = new HashMap();
    while(pa.__len__() != 0) {
      PyTuple po = (PyTuple)pa.pop();
      Object first = po.__finditem__(0)
        .__tojava__(Object.class);
```

```
      Object second = po.__finditem__(1)
        .__tojava__(Object.class);
      map.put(first, second);
    }
    return map;
  }
  /** Turn a Java Map into a PyDictionary,
  suitable for placing into a PythonInterpreter
  @param map The Java Map object
  */
  public static PyDictionary toPyDictionary(Map map) {
    Map m = new HashMap();
    Iterator it = map.entrySet().iterator();
    while(it.hasNext()) {
      Map.Entry e = (Map.Entry)it.next();
      m.put(Py.java2py(e.getKey()),
        Py.java2py(e.getValue()));
    }
    return new PyDictionary(m);
  }
}
```

Here is the unit testing code:

```
// Jython/TestPyUtil.java
import org.python.util.PythonInterpreter;
import java.util.*;

public class TestPyUtil {
  PythonInterpreter pi = new PythonInterpreter();
  public void test1() {
    pi.exec("tup=('fee','fi','fo','fum','fi')");
    List lst = PyUtil.toList(pi, "tup");
    System.out.println(lst);
    System.out.println(new HashSet(lst));
  }
  public void test2() {
    pi.exec("ints=[1,3,5,7,9,11,13,17,19]");
    List lst = PyUtil.toList(pi, "ints");
    System.out.println(lst);
  }
  public void test3() {
    pi.exec("dict = { 1 : 'a', 3 : 'b', " +
      "5 : 'c', 9 : 'd', 11 : 'e'}");
    Map mp = PyUtil.toMap(pi, "dict");
    System.out.println(mp);
  }
  public void test4() {
    Map m = new HashMap();
    m.put("twas", new Integer(11));
    m.put("brillig", new Integer(27));
    m.put("and", new Integer(47));
    m.put("the", new Integer(42));
    m.put("slithy", new Integer(33));
    m.put("toves", new Integer(55));
    System.out.println(m);
    pi.set("m", PyUtil.toPyDictionary(m));
    pi.exec("print(m)");
```

```
    pi.exec("print(m['slithy'])");
  }
  public static void main(String args[]) {
    TestPyUtil test = new TestPyUtil();
    test.test1();
    test.test2();
    test.test3();
    test.test4();
  }
}
```

We'll see the use of the extraction tools in the next section.

## Getting Data Out

There are a number of different ways to extract data from the **PythonInterpreter**. If you simply call the **get()** method, passing it the object identifier as a string, it returns a **PyObject** (part of the **org.python.core** support classes). It's possible to "cast" it using the **__tojava__()** method, but there are better alternatives:

1. The convenience methods in the **Py** class, such as **py2int()**, take a **PyObject** and convert it to a number of different types.
2. An overloaded version of **get()** takes the desired Java **Class** object as a second argument, and produces an object that has that run-time type (so you still need to perform a cast on the result in your Java code).

Using the second approach, getting an array from the **PythonInterpreter** is quite easy. This is especially useful because Python is exceptionally good at manipulating strings and files, and so you will commonly want to extract the results as an array of strings. For example, you can do a wildcard expansion of file names using Python's **glob()**, as shown further down in the following code:

```
// Jython/PythonInterpreterGetting.java
// Getting data from the PythonInterpreter object.
import org.python.util.PythonInterpreter;
import org.python.core.*;
import java.util.*;

public class PythonInterpreterGetting {
  public static void
  main(String[] args) throws PyException  {
    PythonInterpreter interp = new PythonInterpreter();
    interp.exec("a = 100");
    // If you just use the ordinary get(),
    // it returns a PyObject:
    PyObject a = interp.get("a");
    // There's not much you can do with a generic
    // PyObject, but you can print it out:
    System.out.println("a = " + a);
    // If you know the type it's supposed to be,
    // you can "cast" it using __tojava__() to
    // that Java type and manipulate it in Java.
    // To use 'a' as an int, you must use
    // the Integer wrapper class:
    int ai= ((Integer)a.__tojava__(Integer.class))
      .intValue();
    // There are also convenience functions:
    ai = Py.py2int(a);
```

```
    System.out.println("ai + 47 = " + (ai + 47));
    // You can convert it to different types:
    float af = Py.py2float(a);
    System.out.println("af + 47 = " + (af + 47));
    // If you try to cast it to an inappropriate
    // type you'll get a runtime exception:
    //! String as = (String)a.__tojava__(
    //!   String.class);

    // If you know the type, a more useful method
    // is the overloaded get() that takes the
    // desired class as the 2nd argument:
    interp.exec("x = 1 + 2");
    int x = ((Integer)interp
      .get("x", Integer.class)).intValue();
    System.out.println("x = " + x);

    // Since Python is so good at manipulating
    // strings and files, you will often need to
    // extract an array of Strings. Here, a file
    // is read as a Python array:
    interp.exec("lines = " +
      "open('PythonInterpreterGetting.java')" +
      ".readlines()");
    // Pull it in as a Java array of String:
    String[] lines = (String[])
      interp.get("lines", String[].class);
    for(int i = 0; i < 10; i++)
      System.out.print(lines[i]);

    // As an example of useful string tools,
    // global expansion of ambiguous file names
    // using glob is very useful, but it's not
    // part of the standard Jython package, so
    // you'll have to make sure that your
    // Python path is set to include these, or
    // that you deliver the necessary Python
    // files with your application.
    interp.exec("from glob import glob");
    interp.exec("files = glob('*.java')");
    String[] files = (String[])
      interp.get("files", String[].class);
    for(int i = 0; i < files.length; i++)
      System.out.println(files[i]);

    // You can extract tuples and arrays into
    // Java Lists with net.mindview.PyUtil:
    interp.exec("tup = ('fee', 'fi', 'fo', 'fum', 'fi')");
    List tup = PyUtil.toList(interp, "tup");
    System.out.println(tup);
    // It really is a list of String objects:
    System.out.println(tup.get(0).getClass());
    // You can easily convert it to a Set:
    Set tups = new HashSet(tup);
    System.out.println(tups);
    interp.exec("ints=[1,3,5,7,9,11,13,17,19]");
    List ints = PyUtil.toList(interp, "ints");
    System.out.println(ints);
```

```
    // It really is a List of Integer objects:
    System.out.println((ints.get(1)).getClass());

    // If you have a Python dictionary, it can
    // be extracted into a Java Map, again with
    // net.mindview.PyUtil:
    interp.exec("dict = { 1 : 'a', 3 : 'b'," +
      "5 : 'c', 9 : 'd', 11 : 'e' }");
    Map map = PyUtil.toMap(interp, "dict");
    System.out.println("map: " + map);
    // It really is Java objects, not PyObjects:
    Iterator it = map.entrySet().iterator();
    Map.Entry e = (Map.Entry)it.next();
    System.out.println(e.getKey().getClass());
    System.out.println(e.getValue().getClass());
  }
}
```

The last two examples show the extraction of Python tuples and lists into Java **List**s, and Python dictionaries into Java **Map**s. Both of these cases require more processing than is provided in the standard Jython library, so I have again created utilities in **net.mindview.pyton.PyUtil**: **toList()** to produce a **List** from a Python sequence, and **toMap()** to produce a **Map** from a Python dictionary. The **PyUtil** methods make it easier to take important data structures back and forth between Java and Python.

## Multiple Interpreters

It's also worth noting that you can have multiple **PythonInterpreter** objects in a program, and each one has its own name space:

```
// Jython/MultipleJythons.java
// You can run multiple interpreters, each
// with its own name space.
import org.python.util.PythonInterpreter;
import org.python.core.*;

public class MultipleJythons {
  public static void
  main(String[] args) throws PyException  {
    PythonInterpreter
      interp1 =  new PythonInterpreter(),
      interp2 =  new PythonInterpreter();
    interp1.set("a", new PyInteger(42));
    interp2.set("a", new PyInteger(47));
    interp1.exec("print(a)");
    interp2.exec("print(a)");
    PyObject x1 = interp1.get("a");
    PyObject x2 = interp2.get("a");
    System.out.println("a from interp1: " + x1);
    System.out.println("a from interp2: " + x2);
  }
}
```

When you run the program you'll see that the value of **a** is distinct within each **PythonInterpreter**.

## Creating Java classes with Jython

**Note:** Jython 2.5.0 does not support **jythonc**. Support is planned for 2.5.1. **jythonc** basically converted python source to java source, the replacement will generate bytecodes directly, and enable jython code to be imported directly into java (via generated proxies).

Jython can also create Java classes directly from your Jython code. This can produce very useful results, as you are then able to treat the results as if they are native Java classes, albeit with Python power under the hood.

To produce Java classes from Python code, Jython comes with a compiler called **jythonc**.

The process of creating Python classes that will produce Java classes is a bit more complex than when calling Java classes from Python, because the methods in Java classes are statically typed, while Python functions and methods are dynamically typed. Thus, you must somehow tell **jythonc** that a Python method is intended to have a particular set of argument types and that its return value is a particular type. You accomplish this with the **@sig** string, which is placed right after the beginning of the Python method definition (this is the standard location for the Python documentation string). For example:

```
def returnArray(self):
    "@sig public java.lang.String[] returnArray()"
```

The Python definition doesn't specify any return type, but the @sig string gives the full type information about what is being passed and returned. The **jythonc** compiler uses this information to generate the correct Java code.

There's one other set of rules you must follow in order to get a successful compilation: you must inherit from a Java class or interface in your Python class (you do not need to specify the **@sig** signature for methods defined in the superclass/interface). If you do not do this, you won't get your desired methods – unfortunately, **jythonc** gives you no warnings or errors in this case, but you won't get what you want. If you don't see what's missing, it can be very frustrating.

In addition, you must import the appropriate java class and give the correct package specification. In the example below, **java** is imported so you must inherit from **java.lang.Object**, but you could also say **from java.lang import Object** and then you'd just inherit from **Object** without the package specification. Unfortunately, you don't get any warnings or errors if you get this wrong, so you must be patient and keep trying.

Here is an example of a Python class created to produce a Java class. In this case, the Python file is used to build a Java **.class** file, so the class file is the desired target:

```
# Jython/PythonToJavaClass.py
# A Python class converted into a Java class
# Compile with:
# jythonc --package python.java.test PythonToJavaClass.py
from jarray import array
import java

class PythonToJavaClass(java.lang.Object):
    # The '@sig' signature string is used to create the
    # proper signature in the resulting Java code:
    def __init__(self):
        "@sig public PythonToJavaClass()"
        print("Constructor for PythonToJavaClass")

    def simple(self):
```

```
        "@sig public void simple()"
        print("simple()")

    # Returning values to Java:
    def returnString(self):
        "@sig public java.lang.String returnString()"
        return "howdy"

    # You must construct arrays to return along
    # with the type of the array:
    def returnArray(self):
        "@sig public java.lang.String[] returnArray()"
        test = [ "fee", "fi", "fo", "fum" ]
        return array(test, java.lang.String)

    def ints(self):
        "@sig public java.lang.Integer[] ints()"
        test = [ 1, 3, 5, 7, 11, 13, 17, 19, 23 ]
        return array(test, java.lang.Integer)

    def doubles(self):
        "@sig public java.lang.Double[] doubles()"
        test = [ 1, 3, 5, 7, 11, 13, 17, 19, 23 ]
        return array(test, java.lang.Double)

    # Passing arguments in from Java:
    def argIn1(self, a):
        "@sig public void argIn1(java.lang.String a)"
        print("a: %s" % a)
        print("a.__class__", a.__class__)

    def argIn2(self, a):
        "@sig public void argIn1(java.lang.Integer a)"
        print("a + 100: %d" % (a + 100))
        print("a.__class__", a.__class__)

    def argIn3(self, a):
        "@sig public void argIn3(java.util.List a)"
        print("received List:", a, a.__class__)
        print("element type:", a[0].__class__)
        print("a[3] + a[5]:", a[5] + a[7])
        #! print("a[2:5]:", a[2:5]) # Doesn't work

    def argIn4(self, a):
        "@sig public void \
           argIn4(org.python.core.PyArray a)"
        print("received type:", a.__class__)
        print("a: ", a)
        print("element type:", a[0].__class__)
        print("a[3] + a[5]:", a[5] + a[7])
        print("a[2:5]:", a[2:5] # A real Python array)

    # A map must be passed in as a PyDictionary:
    def argIn5(self, m):
        "@sig public void \
           argIn5(org.python.core.PyDictionary m)"
        print("received Map: ", m, m.__class__)
        print("m['3']:", m['3'])
```

```
        for x in m.keys():
            print(x, m[x])
```

First note that **PythonToJavaClass** is inherited from **java.lang.Object**; if you don't do this you will quietly get a Java class without the right signatures. You are not required to inherit from **Object**; any other Java class will do.

This class is designed to demonstrate different arguments and return values, to provide you with enough examples that you'll be able to easily create your own signature strings. The first three of these are fairly self-explanatory, but note the full qualification of the Java name in the signature string.

In **returnArray()**, a Python array must be returned as a Java array. To do this, the Jython **array()** function (from the **jarray** module) must be used, along with the type of the class for the resulting array. Any time you need to return an array to Java, you must use **array()**, as seen in the methods **ints()** and **doubles()**.

The last methods show how to pass arguments in from Java. Basic types happen automatically as long as you specify them in the **@sig** string, but you must use objects and you cannot pass in primitives (that is, primitives must be ensconced in wrapper objects, such as **Integer**).

In **argIn3()**, you can see that a Java **List** is transparently converted to something that behaves just like a Python array, but is not a true array because you cannot take a slice from it. If you want a true Python array, then you must create and pass a **PyArray** as in **argIn4()**, where the slice is successful. Similarly, a Java **Map** must come in as a **PyDictionary** in order to be treated as a Python dictionary.

Here is the Java program to exercise the Java classes produced by the above Python code. You can't compile **TestPythonToJavaClass.java** until **PythonToJavaClass.class** is available:

```
// Jython/TestPythonToJavaClass.java
import java.lang.reflect.*;
import java.util.*;
import org.python.core.*;
import java.util.*;
import net.mindview.python.*;
// The package with the Python-generated classes:
import python.java.test.*;

public class TestPythonToJavaClass {
  PythonToJavaClass p2j = new PythonToJavaClass();
  public void testDumpClassInfo() {
    System.out.println(
      Arrays.toString(
        p2j.getClass().getConstructors()));
    Method[] methods = p2j.getClass().getMethods();
    for(int i = 0; i < methods.length; i++) {
      String nm = methods[i].toString();
      if(nm.indexOf("PythonToJavaClass") != -1)
        System.out.println(nm);
    }
  }
  public static void main(String[] args) {
    p2j.simple();
    System.out.println(p2j.returnString());
    System.out.println(
      Arrays.toString(p2j.returnArray()));
    System.out.println(
      Arrays.toString(p2j.ints());
    System.out.println(
      Arrays.toString(p2j.doubles()));
    p2j.argIn1("Testing argIn1()");
```

```
    p2j.argIn2(new Integer(47));
    ArrayList a = new ArrayList();
    for(int i = 0; i < 10; i++)
      a.add(new Integer(i));
    p2j.argIn3(a);
    p2j.argIn4(
      new PyArray(Integer.class, a.toArray()));
    Map m = new HashMap();
    for(int i = 0; i < 10; i++)
      m.put("" + i, new Float(i));
    p2j.argIn5(PyUtil.toPyDictionary(m));
  }
}
```

For Python support, you'll usually only need to import the classes in **org.python.core**. Everything else in the above example is fairly straightforward, as **PythonToJavaClass** appears, from the Java side, to be just another Java class. **dumpClassInfo()** uses reflection to verify that the method signatures specified in **PythonToJavaClass.py** have come through properly.

## Building Java Classes from Python

Part of the trick of creating Java classes from Python code is the @sig information in the method documentation strings. But there's a second problem which stems from the fact that Python has no "package" keyword – the Python equivalent of packages (modules) are implicitly created based on the file name. However, to bring the resulting class files into the Java program, **jythonc** must be given information about how to create the Java package for the Python code. This is done on the **jythonc** command line using the **–package** flag, followed by the package name you wish to produce (including the separation dots, just as you would give the package name using the **package** keyword in a Java program). This will put the resulting **.class** files in the appropriate subdirectory off of the current directory. Then you only need to import the package in your Java program, as shown above (you'll need '.' in your CLASSPATH in order to run it from the code directory).

Here are the **make** dependency rules that I used to build the above example (the backslashes at the ends of the lines are understood by **make** to be line continuations):

```
TestPythonToJavaClass.class: \\
        TestPythonToJavaClass.java \\
        python\java\test\PythonToJavaClass.class
    javac TestPythonToJavaClass.java

python\java\test\PythonToJavaClass.class: \\
        PythonToJavaClass.py
    jythonc.bat --package python.java.test \\
    PythonToJavaClass.py
```

The first target, **TestPythonToJavaClass.class**, depends on both **TestPythonToJavaClass.java** and the **PythonToJavaClass.class**, which is the Python code that's converted to a class file. This latter, in turn, depends on the Python source code. Note that it's important that the directory where the target lives be specified, so that the makefile will create the Java program with the minimum necessary amount of rebuilding.

## Summary

This chapter has arguably gone much deeper into Jython than required to use the interpreter design pattern. Indeed, once you decide that you need to use interpreter and that you're not going to get lost inventing your own language, the solution of installing Jython is quite simple, and you can at least get started by following the **GreenHouseController** example.

Of course, that example is often too simple and you may need something more sophisticated, often requiring more interesting data to be passed back and forth. When I encountered the limited documentation, I felt it necessary to come up with a more thorough examination of Jython.

In the process, note that there could be another equally powerful design pattern lurking in here, which could perhaps be called *multiple languages* or *language hybridizing*. This is based on the experience of having each language solve a certain class of problems better than the other; by combining languages you can solve problems much faster than with either language by itself. CORBA is another way to bridge across languages, and at the same time bridging between computers and operating systems.

To me, Python and Java present a very potent combination for program development because of Java's architecture and tool set, and Python's extremely rapid development (generally considered to be 5-10 times faster than C++ or Java). Python is usually slower, however, but even if you end up re-coding parts of your program for speed, the initial fast development will allow you to more quickly flesh out the system and uncover and solve the critical sections. And often, the execution speed of Python is not a problem – in those cases it's an even bigger win. A number of commercial products already use Java and Jython, and because of the terrific productivity leverage I expect to see this happen more in the future.

## Exercises

1. Modify **GreenHouseLanguage.py** so that it checks the times for the events and runs those events at the appropriate times.
2. Modify **GreenHouseLanguage.py** so that it calls a function for **action** instead of just printing a string.
3. Create a Swing application with a **JTextField** (where the user will enter commands) and a **JTextArea** (where the command results will be displayed). Connect to a **PythonInterpreter** object so that the output will be sent to the **JTextArea** (which should scroll). You'll need to locate the **PythonInterpreter** command that redirects the output to a Java stream.
4. Modify **GreenHouseLanguage.py** to add a master controller class (instead of the static array inside **Event**) and provide a **run()** method for each of the subclasses. Each **run()** should create and use an object from the standard Java library during its execution. Modify **GreenHouseController.java** to use this new class.
5. Modify the resulting **GreenHouseLanguage.py** from exercise two to produce Java classes (add the @sig documentation strings to produce the correct Java signatures, and create a makefile to build the Java **.class** files). Write a Java program that uses these classes.
6. Modify **GreenHouseLanguage.py** so that the subclasses of **Event** are not discrete classes, but are instead *generated* by a single function which creates the class and the associated string dynamically.

CHAPTER 20

# Part II: Idioms

CHAPTER 21

# Discovering the Details About Your Platform

The Python library XXX will give you some information about your machine, but it falls short. Here's a rather messy, but useful way to figure out everything else.

Just a starting point:

```
# MachineDiscovery/detect_CPUs.py
def detect_CPUs():
    """
    Detects the number of CPUs on a system. Cribbed from pp.
    """
    # Linux, Unix and MacOS:
    if hasattr(os, "sysconf"):
        if os.sysconf_names.has_key("SC_NPROCESSORS_ONLN"):
            # Linux & Unix:
            ncpus = os.sysconf("SC_NPROCESSORS_ONLN")
            if isinstance(ncpus, int) and ncpus > 0:
                return ncpus
        else: # OSX:
            return int(os.popen2("sysctl -n hw.ncpu")[1].read())
    # Windows:
    if os.environ.has_key("NUMBER_OF_PROCESSORS"):
            ncpus = int(os.environ["NUMBER_OF_PROCESSORS"]);
            if ncpus > 0:
                return ncpus
    return 1 # Default
```

CHAPTER 22

# A Canonical Form for Command-Line Programs

Creating Python programs for command-line use involves a certain amount of repetitious coding, which can often be left off or forgotten. Here is a form which includes everything.

Note that if you are using Windows, you can add Python programs to your "File New" menu and automatically include the above text in the new file. This article shows you how. Other operating systems have their own automation features.

CHAPTER 23

# Messenger/Data Transfer Object

The *Messenger* or *Data Transfer Object* is a way to pass a clump of information around. The most typical place for this is in return values from functions, where tuples or dictionaries are often used. However, those rely on indexing; in the case of tuples this requires the consumer to keep track of numerical order, and in the case of a **dict** you must use the `d["name"]` syntax which can be slightly less desireable.

A Messenger is simply an object with attributes corresponding to the names of the data you want to pass around or return:

```
# Messenger/MessengerIdiom.py

class Messenger:
    def __init__(self, **kwargs):
        self.__dict__ = kwargs

m = Messenger(info="some information", b=['a', 'list'])
m.more = 11
print m.info, m.b, m.more
```

The trick here is that the `__dict__` for the object is just assigned to the **dict** that is automatically created by the `**kwargs` argument.

Although one could easily create a `Messenger` class and put it into a library and import it, there are so few lines to describe it that it usually makes more sense to just define it in-place whenever you need it – it is probably easier for the reader to follow, as well.

CHAPTER 24

# Part III: Patterns

CHAPTER 25

# The Pattern Concept

"Design patterns help you learn from others' successes instead of your own failures[1]."

Probably the most important step forward in object-oriented design is the "design patterns" movement, chronicled in *Design Patterns (ibid)*[2]. That book shows 23 different solutions to particular classes of problems. In this book, the basic concepts of design patterns will be introduced along with examples. This should whet your appetite to read *Design Patterns* by Gamma, et. al., a source of what has now become an essential, almost mandatory, vocabulary for OOP programmers.

The latter part of this book contains an example of the design evolution process, starting with an initial solution and moving through the logic and process of evolving the solution to more appropriate designs. The program shown (a trash sorting simulation) has evolved over time, and you can look at that evolution as a prototype for the way your own design can start as an adequate solution to a particular problem and evolve into a flexible approach to a class of problems.

## What is a Pattern?

Initially, you can think of a pattern as an especially clever and insightful way of solving a particular class of problems. That is, it looks like a lot of people have worked out all the angles of a problem and have come up with the most general, flexible solution for it. The problem could be one you have seen and solved before, but your solution probably didn't have the kind of completeness you'll see embodied in a pattern.

Although they're called "design patterns," they really aren't tied to the realm of design. A pattern seems to stand apart from the traditional way of thinking about analysis, design, and implementation. Instead, a pattern embodies a complete idea within a program, and thus it can sometimes appear at the analysis phase or high-level design phase. This is interesting because a pattern has a direct implementation in code and so you might not expect it to show up before low-level design or implementation (and in fact you might not realize that you need a particular pattern until you get to those phases).

The basic concept of a pattern can also be seen as the basic concept of program design: adding a layer of abstraction. Whenever you abstract something you're isolating particular details, and one of the most

[1] From Mark Johnson.

[2] But be warned: the examples are in C++.

compelling motivations behind this is to *separate things that change from things that stay the same*. Another way to put this is that once you find some part of your program that's likely to change for one reason or another, you'll want to keep those changes from propagating other changes throughout your code. Not only does this make the code much cheaper to maintain, but it also turns out that it is usually simpler to understand (which results in lowered costs).

Often, the most difficult part of developing an elegant and cheap-to-maintain design is in discovering what I call "the vector of change." (Here, "vector" refers to the maximum gradient and not a container class.) This means finding the most important thing that changes in your system, or put another way, discovering where your greatest cost is. Once you discover the vector of change, you have the focal point around which to structure your design.

So the goal of design patterns is to isolate changes in your code. If you look at it this way, you've been seeing some design patterns already in this book. For example, inheritance can be thought of as a design pattern (albeit one implemented by the compiler). It allows you to express differences in behavior (that's the thing that changes) in objects that all have the same interface (that's what stays the same). Composition can also be considered a pattern, since it allows you to change-dynamically or statically-the objects that implement your class, and thus the way that class works.

Another pattern that appears in *Design Patterns* is the *iterator*, which has been implicitly available in **for** loops from the beginning of the language, and was introduced as an explicit feature in Python 2.2. An iterator allows you to hide the particular implementation of the container as you're stepping through and selecting the elements one by one. Thus, you can write generic code that performs an operation on all of the elements in a sequence without regard to the way that sequence is built. Thus your generic code can be used with any object that can produce an iterator.

## Classifying Patterns

The *Design Patterns* book discusses 23 different patterns, classified under three purposes (all of which revolve around the particular aspect that can vary). The three purposes are:

1. **Creational**: how an object can be created. This often involves isolating the details of object creation so your code isn't dependent on what types of objects there are and thus doesn't have to be changed when you add a new type of object. The aforementioned *Singleton* is classified as a creational pattern, and later in this book you'll see examples of *Factory Method* and *Prototype*.
2. **Structural**: designing objects to satisfy particular project constraints. These work with the way objects are connected with other objects to ensure that changes in the system don't require changes to those connections.
3. **Behavioral**: objects that handle particular types of actions within a program. These encapsulate processes that you want to perform, such as interpreting a language, fulfilling a request, moving through a sequence (as in an iterator), or implementing an algorithm. This book contains examples of the *Observer* and the *Visitor* patterns.

The *Design Patterns* book has a section on each of its 23 patterns along with one or more examples for each, typically in C++ but sometimes in Smalltalk. (You'll find that this doesn't matter too much since you can easily translate the concepts from either language into Python.) This book will not repeat all the patterns shown in *Design Patterns* since that book stands on its own and should be studied separately. Instead, this book will give some examples that should provide you with a decent feel for what patterns are about and why they are so important.

After years of looking at these things, it began to occur to me that the patterns themselves use basic principles of organization, other than (and more fundamental than) those described in *Design Patterns*. These principles are based on the structure of the implementations, which is where I have seen great similarities

between patterns (more than those expressed in *Design Patterns*). Although we generally try to avoid implementation in favor of interface, I have found that it's often easier to think about, and especially to learn about, the patterns in terms of these structural principles. This book will attempt to present the patterns based on their structure instead of the categories presented in *Design Patterns*.

## Pattern Taxonomy

One of the events that's occurred with the rise of design patterns is what could be thought of as the "pollution" of the term - people have begun to use the term to mean just about anything synonymous with "good." After some pondering, I've come up with a sort of hierarchy describing a succession of different types of categories:

1. **Idiom**: how we write code in a particular language to do this particular type of thing. This could be something as common as the way that you code the process of stepping through an array in C (and not running off the end).
2. **Specific Design**: the solution that we came up with to solve this particular problem. This might be a clever design, but it makes no attempt to be general.
3. **Standard Design**: a way to solve this *kind* of problem. A design that has become more general, typically through reuse.
4. **Design Pattern**: how to solve an entire class of similar problem. This usually only appears after applying a standard design a number of times, and then seeing a common pattern throughout these applications.

I feel this helps put things in perspective, and to show where something might fit. However, it doesn't say that one is better than another. It doesn't make sense to try to take every problem solution and generalize it to a design pattern - it's not a good use of your time, and you can't force the discovery of patterns that way; they tend to be subtle and appear over time.

One could also argue for the inclusion of *Analysis Pattern* and *Architectural Pattern* in this taxonomy.

## Design Structures

One of the struggles that I've had with design patterns is their classification - I've often found the GoF approach to be too obscure, and not always very helpful. Certainly, the *Creational* patterns are fairly straightforward: how are you going to create your objects? This is a question you normally need to ask, and the name brings you right to that group of patterns. But I find *Structural* and *Behavioral* to be far less useful distinctions. I have not been able to look at a problem and say "clearly, you need a structural pattern here," so that classification doesn't lead me to a solution (I'll readily admit that I may be missing something here).

I've labored for awhile with this problem, first noting that the underlying structure of some of the GoF patterns are similar to each other, and trying to develop relationships based on that similarity. While this was an interesting experiment, I don't think it produced much of use in the end because the point is to solve problems, so a helpful approach will look at the problem to solve and try to find relationships between the problem and potential solutions.

To that end, I've begun to try to collect basic design structures, and to try to see if there's a way to relate those structures to the various design patterns that appear in well thought-out systems. Currently, I'm just trying to make a list, but eventually I hope to make steps towards connecting these structures with patterns (or I may come up with a different approach altogether - this is still in its formative stages).

Here[3] is the present list of candidates, only some of which will make it to the final list. Feel free to suggest

[3] This list includes suggestions by Kevlin Henney, David Scott, and others.

others, or possibly relationships with patterns.

- **Encapsulation**: self containment and embodying a model of usage
- **Gathering**
- **Localization**
- **Separation**
- **Hiding**
- **Guarding**
- **Connector**
- **Barrier/fence**
- **Variation in behavior**
- **Notification**
- **Transaction**
- **Mirror: "the ability to keep a parallel universe(s) in step with the** golden world"
- **Shadow: "follows your movement and does something different in a different** medium" (May be a variation on Proxy).

## Design Principles

When I put out a call for ideas in my newsletter[4], a number of suggestions came back which turned out to be very useful, but different than the above classification, and I realized that a list of design principles is at least as important as design structures, but for a different reason: these allow you to ask questions about your proposed design, to apply tests for quality.

- **Principle of least astonishment** (don't be astonishing).
- **Make common things easy, and rare things possible**
- **Consistency**. One thing has become very clear to me, especially because of Python: the more random rules you pile onto the programmer, rules that have nothing to do with solving the problem at hand, the slower the programmer can produce. And this does not appear to be a linear factor, but an exponential one.
- **Law of Demeter**: a.k.a. "Don't talk to strangers." An object should only reference itself, its attributes, and the arguments of its methods. This may also be a way to say "minimize coupling."
- **Independence** or **Orthogonality**. Express independent ideas independently. This complements Separation, Encapsulation and Variation, and is part of the Low-Coupling-High-Cohesion message.
- **Managed Coupling**. Simply stating that we should have "low coupling" in a design is usually too vague - coupling happens, and the important issue is to acknowledge it and control it, to say "coupling can cause problems" and to compensate for those problems with a well-considered design or pattern.
- **Subtraction**: a design is finished when you cannot take anything else away[5].

[4] A free email publication. See *www.BruceEckel.com* to subscribe.

[5] This idea is generally attributed to Antoine de St. Exupery from *The Little Prince*: "La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever," or: "perfection is reached not when there's nothing left to add, but when there's nothing left to remove".

- **Simplicity before generality**[6]. (A variation of *Occam's Razor*, which says "the simplest solution is the best"). A common problem we find in frameworks is that they are designed to be general purpose without reference to actual systems. This leads to a dizzying array of options that are often unused, misused or just not useful. However, most developers work on specific systems, and the quest for generality does not always serve them well. The best route to generality is through understanding well-defined specific examples. So, this principle acts as the tie breaker between otherwise equally viable design alternatives. Of course, it is entirely possible that the simpler solution is the more general one.
- **Reflexivity** (my suggested term). One abstraction per class, one class per abstraction. Might also be called Isomorphism.
- **Once and once only**: Avoid duplication of logic and structure where the duplication is not accidental, ie where both pieces of code express the same intent for the same reason.

In the process of brainstorming this idea, I hope to come up with a small handful of fundamental ideas that can be held in your head while you analyze a problem. However, other ideas that come from this list may end up being useful as a checklist while walking through and analyzing your design.

## Further Reading

Alex Martelli's Video Lectures on Design Patterns in Python: http://www.catonmat.net/blog/learning-python-design-patterns-through-video-lectures/

[6] From an email from Kevlin Henney.

# CHAPTER 26

## The Singleton

Possibly the simplest design pattern is the *singleton*, which is a way to provide one and only one object of a particular type. To accomplish this, you must take control of object creation out of the hands of the programmer. One convenient way to do this is to delegate to a single instance of a private nested inner class:

```
# Singleton/SingletonPattern.py

class OnlyOne:
    class __OnlyOne:
        def __init__(self, arg):
            self.val = arg
        def __str__(self):
            return repr(self) + self.val
    instance = None
    def __init__(self, arg):
        if not OnlyOne.instance:
            OnlyOne.instance = OnlyOne.__OnlyOne(arg)
        else:
            OnlyOne.instance.val = arg
    def __getattr__(self, name):
        return getattr(self.instance, name)

x = OnlyOne('sausage')
print(x)
y = OnlyOne('eggs')
print(y)
z = OnlyOne('spam')
print(z)
print(x)
print(y)
print(`x`)
print(`y`)
print(`z`)
output = '''
<__main__.__OnlyOne instance at 0076B7AC>sausage
```

```
<__main__.__OnlyOne instance at 0076B7AC>eggs
<__main__.__OnlyOne instance at 0076B7AC>spam
<__main__.__OnlyOne instance at 0076B7AC>spam
<__main__.__OnlyOne instance at 0076B7AC>spam
<__main__.OnlyOne instance at 0076C54C>
<__main__.OnlyOne instance at 0076DAAC>
<__main__.OnlyOne instance at 0076AA3C>
'''
```

Because the inner class is named with a double underscore, it is private so the user cannot directly access it. The inner class contains all the methods that you would normally put in the class if it weren't going to be a singleton, and then it is wrapped in the outer class which controls creation by using its constructor. The first time you create an **OnlyOne**, it initializes **instance**, but after that it just ignores you.

Access comes through delegation, using the **__getattr__( )** method to redirect calls to the single instance. You can see from the output that even though it appears that multiple objects have been created, the same **__OnlyOne** object is used for both. The instances of **OnlyOne** are distinct but they all proxy to the same **__OnlyOne** object.

Note that the above approach doesn't restrict you to creating only one object. This is also a technique to create a limited pool of objects. In that situation, however, you can be confronted with the problem of sharing objects in the pool. If this is an issue, you can create a solution involving a check-out and check- in of the shared objects.

A variation on this technique uses the class method **__new__** added in Python 2.2:

```
# Singleton/NewSingleton.py

class OnlyOne(object):
    class __OnlyOne:
        def __init__(self):
            self.val = None
        def __str__(self):
            return `self` + self.val
    instance = None
    def __new__(cls): # __new__ always a classmethod
        if not OnlyOne.instance:
            OnlyOne.instance = OnlyOne.__OnlyOne()
        return OnlyOne.instance
    def __getattr__(self, name):
        return getattr(self.instance, name)
    def __setattr__(self, name):
        return setattr(self.instance, name)

x = OnlyOne()
x.val = 'sausage'
print(x)
y = OnlyOne()
y.val = 'eggs'
print(y)
z = OnlyOne()
z.val = 'spam'
print(z)
print(x)
print(y)
#<hr>
output = '''
<__main__.__OnlyOne instance at 0x00798900>sausage
```

```
<__main__.__OnlyOne instance at 0x00798900>eggs
<__main__.__OnlyOne instance at 0x00798900>spam
<__main__.__OnlyOne instance at 0x00798900>spam
<__main__.__OnlyOne instance at 0x00798900>spam
'''
```

Alex Martelli makes the observation that what we really want with a Singleton is to have a single set of state data for all objects. That is, you could create as many objects as you want and as long as they all refer to the same state information then you achieve the effect of Singleton. He accomplishes this with what he calls the *Borg*[1], which is accomplished by setting all the **__dict__**s to the same static piece of storage:

```
# Singleton/BorgSingleton.py
# Alex Martelli's 'Borg'

class Borg:
    _shared_state = {}
    def __init__(self):
        self.__dict__ = self._shared_state

class Singleton(Borg):
    def __init__(self, arg):
        Borg.__init__(self)
        self.val = arg
    def __str__(self): return self.val

x = Singleton('sausage')
print(x)
y = Singleton('eggs')
print(y)
z = Singleton('spam')
print(z)
print(x)
print(y)
print(`x`)
print(`y`)
print(`z`)
output = '''
sausage
eggs
spam
spam
spam
<__main__.Singleton instance at 0079EF2C>
<__main__.Singleton instance at 0079E10C>
<__main__.Singleton instance at 00798F9C>
'''
```

This has an identical effect as **SingletonPattern.py** does, but it's more elegant. In the former case, you must wire in *Singleton* behavior to each of your classes, but *Borg* is designed to be easily reused through inheritance.

A simpler version[2] of this takes advantage of the fact that there's only one instance of a class variable:

```
# Singleton/ClassVariableSingleton.py
class SingleTone(object):
```

[1] From the television show *Star Trek: The Next Generation*. The Borg are a hive-mind collective: "we are all one."
[2] From Dmitry Balabanov.

```
    __instance = None
    def __new__(cls, val):
        if SingleTone.__instance is None:
            SingleTone.__instance = object.__new__(cls)
        SingleTone.__instance.val = val
        return SingleTone.__instance
```

Two other interesting ways to define singleton[3] include wrapping a class and using metaclasses. The first approach could be thought of as a *class decorator* (decorators will be defined later in the book), because it takes the class of interest and adds functionality to it by wrapping it in another class:

```
# Singleton/SingletonDecorator.py
class SingletonDecorator:
    def __init__(self,klass):
        self.klass = klass
        self.instance = None
    def __call__(self,*args,**kwds):
        if self.instance == None:
            self.instance = self.klass(*args,**kwds)
        return self.instance

class foo: pass
foo = SingletonDecorator(foo)

x=foo()
y=foo()
z=foo()
x.val = 'sausage'
y.val = 'eggs'
z.val = 'spam'
print(x.val)
print(y.val)
print(z.val)
print(x is y is z)
```

[[ Description ]]

The second approach uses metaclasses, a topic I do not yet understand but which looks very interesting and powerful indeed (note that Python 2.2 has improved/simplified the metaclass syntax, and so this example may change):

```
# Singleton/SingletonMetaClass.py
class SingletonMetaClass(type):
    def __init__(cls,name,bases,dict):
        super(SingletonMetaClass,cls)\
          .__init__(name,bases,dict)
        original_new = cls.__new__
        def my_new(cls,*args,**kwds):
            if cls.instance == None:
                cls.instance = \
                  original_new(cls,*args,**kwds)
            return cls.instance
        cls.instance = None
        cls.__new__ = staticmethod(my_new)

class bar(object):
```

[3] Suggested by Chih-Chung Chang.

```
    __metaclass__ = SingletonMetaClass
    def __init__(self,val):
        self.val = val
    def __str__(self):
        return `self` + self.val

x=bar('sausage')
y=bar('eggs')
z=bar('spam')
print(x)
print(y)
print(z)
print(x is y is z)
```

[[ Long, detailed, informative description of what metaclasses are and how they work, magically inserted here ]]

## Exercises

1. **SingletonPattern.py** always creates an object, even if it's never used. Modify this program to use *lazy initialization*, so the singleton object is only created the first time that it is needed.
2. Using **SingletonPattern.py** as a starting point, create a class that manages a fixed number of its own objects. Assume the objects are database connections and you only have a license to use a fixed quantity of these at any one time.
3. Modify **BorgSingleton.py** so that it uses a class **__new__( )** method.

CHAPTER 27

# Building Application Frameworks

An application framework allows you to inherit from a class or set of classes and create a new application, reusing most of the code in the existing classes and overriding one or more methods in order to customize the application to your needs. A fundamental concept in the application framework is the *Template Method* which is typically hidden beneath the covers and drives the application by calling the various methods in the base class (some of which you have overridden in order to create the application).

For example, whenever you create an applet you're using an application framework: you inherit from **JApplet** and then override **init( )**. The applet mechanism (which is a *Template Method*) does the rest by drawing the screen, handling the event loop, resizing, etc.

## Template Method

An important characteristic of the *Template Method* is that it is defined in the base class and cannot be changed. It's sometimes a **private** method but it's virtually always **final**. It calls other base-class methods (the ones you override) in order to do its job, but it is usually called only as part of an initialization process (and thus the client programmer isn't necessarily able to call it directly):

```
# AppFrameworks/TemplateMethod.py
# Simple demonstration of Template Method.

class ApplicationFramework:
    def __init__(self):
        self.__templateMethod()
    def __templateMethod(self):
        for i in range(5):
            self.customize1()
            self.customize2()

# Create an "application":
class MyApp(ApplicationFramework):
    def customize1(self):
        print("Nudge, nudge, wink, wink! ",)
```

```
    def customize2(self):
        print("Say no more, Say no more!")

MyApp()
```

The base-class constructor is responsible for performing the necessary initialization and then starting the "engine" (the template method) that runs the application (in a GUI application, this "engine" would be the main event loop). The client programmer simply provides definitions for **customize1( )** and **customize2( )** and the "application" is ready to run.

We'll see *Template Method* numerous other times throughout the book.

## Exercises

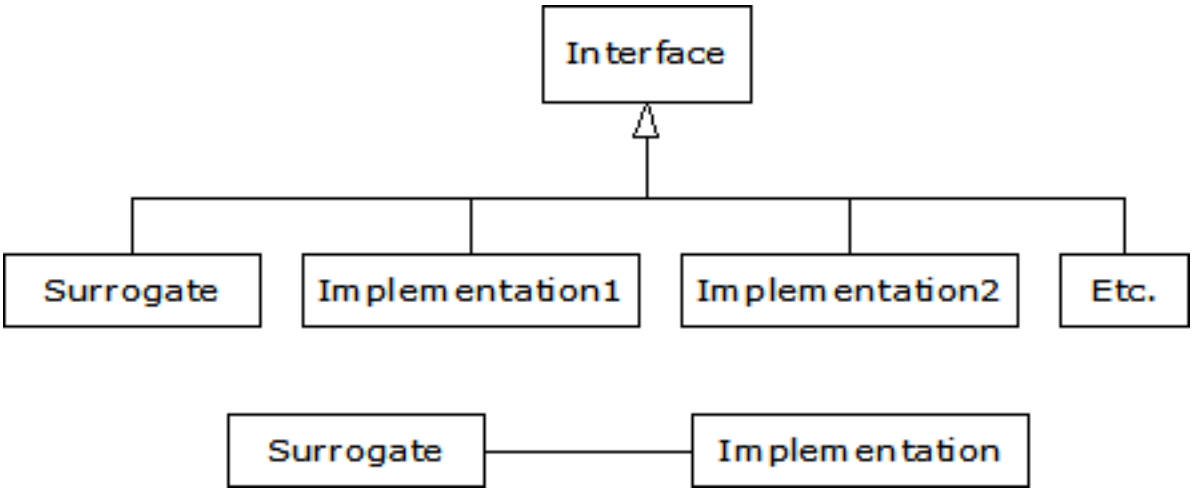
1. Create a framework that takes a list of file names on the command line. It opens each file except the last for reading, and the last for writing. The framework will process each input file using an undetermined policy and write the output to the last file. Inherit to customize this framework to create two separate applications:
   (a) Converts all the letters in each file to uppercase.
   (b) Searches the files for words given in the first file.

CHAPTER 28

# Fronting for an Implementation

Both *Proxy* and *State* provide a surrogate class that you use in your code; the real class that does the work is hidden behind this surrogate class. When you call a method in the surrogate, it simply turns around and calls the method in the implementing class. These two patterns are so similar that the *Proxy* is simply a special case of *State*. One is tempted to just lump the two together into a pattern called *Surrogate*, but the term "proxy" has a long-standing and specialized meaning, which probably explains the reason for the two different patterns.

The basic idea is simple: from a base class, the surrogate is derived along with the class or classes that provide the actual implementation:




When a surrogate object is created, it is given an implementation to which to send all of the method calls.

Structurally, the difference between *Proxy* and *State* is simple: a *Proxy* has only one implementation, while *State* has more than one. The application of the patterns is considered (in *Design Patterns*) to be distinct: *Proxy* is used to control access to its implementation, while *State* allows you to change the implementation dynamically. However, if you expand your notion of "controlling access to implementation" then the two fit neatly together.

## Proxy

If we implement *Proxy* by following the above diagram, it looks like this:

```
# Fronting/ProxyDemo.py
# Simple demonstration of the Proxy pattern.

class Implementation:
    def f(self):
        print("Implementation.f()")
    def g(self):
        print("Implementation.g()")
    def h(self):
        print("Implementation.h()")

class Proxy:
    def __init__(self):
        self.__implementation = Implementation()
    # Pass method calls to the implementation:
    def f(self): self.__implementation.f()
    def g(self): self.__implementation.g()
    def h(self): self.__implementation.h()

p = Proxy()
p.f(); p.g(); p.h()
```

It isn't necessary that **Implementation** have the same interface as **Proxy**; as long as **Proxy** is somehow "speaking for" the class that it is referring method calls to then the basic idea is satisfied (note that this statement is at odds with the definition for Proxy in GoF). However, it is convenient to have a common interface so that **Implementation** is forced to fulfill all the methods that **Proxy** needs to call.

Of course, in Python we have a delegation mechanism built in, so it makes the **Proxy** even simpler to implement:

```
# Fronting/ProxyDemo2.py
# Simple demonstration of the Proxy pattern.

class Implementation2:
    def f(self):
        print("Implementation.f()")
    def g(self):
        print("Implementation.g()")
    def h(self):
        print("Implementation.h()")

class Proxy2:
    def __init__(self):
        self.__implementation = Implementation2()
    def __getattr__(self, name):
        return getattr(self.__implementation, name)

p = Proxy2()
p.f(); p.g(); p.h();
```

The beauty of using **__getattr__( )** is that **Proxy2** is completely generic, and not tied to any particular implementation (in Java, a rather complicated "dynamic proxy" has been invented to accomplish this same thing).

## State

The *State* pattern adds more implementations to *Proxy*, along with a way to switch from one implementation to another during the lifetime of the surrogate:

```
# Fronting/StateDemo.py
# Simple demonstration of the State pattern.

class State_d:
    def __init__(self, imp):
        self.__implementation = imp
    def changeImp(self, newImp):
        self.__implementation = newImp
    # Delegate calls to the implementation:
    def __getattr__(self, name):
        return getattr(self.__implementation, name)

class Implementation1:
    def f(self):
        print("Fiddle de dum, Fiddle de dee,")
    def g(self):
        print("Eric the half a bee.")
    def h(self):
        print("Ho ho ho, tee hee hee,")

class Implementation2:
    def f(self):
        print("We're Knights of the Round Table.")
    def g(self):
        print("We dance whene'er we're able.")
    def h(self):
        print("We do routines and chorus scenes")

def run(b):
    b.f()
    b.g()
    b.h()
    b.g()

b = State_d(Implementation1())
run(b)
b.changeImp(Implementation2())
run(b)
```

You can see that the first implementation is used for a bit, then the second implementation is swapped in and that is used.

The difference between *Proxy* and *State* is in the problems that are solved. The common uses for *Proxy* as described in *Design Patterns* are:

1. **Remote proxy**. This proxies for an object in a different address space. A remote proxy is created for you automatically by the RMI compiler **rmic** as it creates stubs and skeletons.
2. **Virtual proxy**. This provides "lazy initialization" to create expensive objects on demand.
3. **Protection proxy**. Used when you don't want the client programmer to have full access to the proxied object.
4. **Smart reference**. To add additional actions when the proxied object is accessed. For example, or to

keep track of the number of references that are held for a particular object, in order to implement the *copy-on-write* idiom and prevent object aliasing. A simpler example is keeping track of the number of calls to a particular method.

You could look at a Python reference as a kind of protection proxy, since it controls access to the actual object on the heap (and ensures, for example, that you don't use a **null** reference).

[[ Rewrite this: In *Design Patterns*, *Proxy* and *State* are not seen as related to each other because the two are given (what I consider arbitrarily) different structures. *State*, in particular, uses a separate implementation hierarchy but this seems to me to be unnecessary unless you have decided that the implementation is not under your control (certainly a possibility, but if you own all the code there seems to be no reason not to benefit from the elegance and helpfulness of the single base class). In addition, *Proxy* need not use the same base class for its implementation, as long as the proxy object is controlling access to the object it "fronting" for. Regardless of the specifics, in both *Proxy* and *State* a surrogate is passing method calls through to an implementation object.]]]

CHAPTER 29

# StateMachine

While *State* has a way to allow the client programmer to change the implementation, *StateMachine* imposes a structure to automatically change the implementation from one object to the next. The current implementation represents the state that a system is in, and the system behaves differently from one state to the next (because it uses *State*). Basically, this is a "state machine" using objects.

The code that moves the system from one state to the next is often a *Template Method*, as seen in the following framework for a basic state machine.

Each state can be **run( )** to perform its behavior, and (in this design) you can also pass it an "input" object so it can tell you what new state to move to based on that "input". The key distinction between this design and the next is that here, each **State** object decides what other states it can move to, based on the "input", whereas in the subsequent design all of the state transitions are held in a single table. Another way to put it is that here, each **State** object has its own little **State** table, and in the subsequent design there is a single master state transition table for the whole system:

```
# StateMachine/State.py
# A State has an operation, and can be moved
# into the next State given an Input:

class State:
    def run(self):
        assert 0, "run not implemented"
    def next(self, input):
        assert 0, "next not implemented"
```

This class is clearly unnecessary, but it allows us to say that something is a **State** object in code, and provide a slightly different error message when all the methods are not implemented. We could have gotten basically the same effect by saying:

```
class State: pass
```

because we would still get exceptions if **run( )** or **next( )** were called for a derived type, and they hadn't been implemented.

The **StateMachine** keeps track of the current state, which is initialized by the constructor. The **runAll( )** method takes a list of **Input** objects. This method not only moves to the next state, but it also calls **run( )** for each state object - thus you can see it's an expansion of the idea of the **State** pattern, since **run( )** does something different depending on the state that the system is in:

```
# StateMachine/StateMachine.py
# Takes a list of Inputs to move from State to
# State using a template method.

class StateMachine:
    def __init__(self, initialState):
        self.currentState = initialState
        self.currentState.run()
    # Template method:
    def runAll(self, inputs):
        for i in inputs:
            print(i)
            self.currentState = self.currentState.next(i)
            self.currentState.run()
```

I've also treated **runAll( )** as a template method. This is typical, but certainly not required - you could concievably want to override it, but typically the behavior change will occur in **State**'s **run( )** instead.

At this point the basic framework for this style of *StateMachine* (where each state decides the next states) is complete. As an example, I'll use a fancy mousetrap that can move through several states in the process of trapping a mouse[1]. The mouse classes and information are stored in the **mouse** package, including a class representing all the possible moves that a mouse can make, which will be the inputs to the state machine:

```
# StateMachine/mouse/MouseAction.py

class MouseAction:
    def __init__(self, action):
        self.action = action
    def __str__(self): return self.action
    def __cmp__(self, other):
        return cmp(self.action, other.action)
    # Necessary when __cmp__ or __eq__ is defined
    # in order to make this class usable as a
    # dictionary key:
    def __hash__(self):
        return hash(self.action)

# Static fields; an enumeration of instances:
MouseAction.appears = MouseAction("mouse appears")
MouseAction.runsAway = MouseAction("mouse runs away")
MouseAction.enters = MouseAction("mouse enters trap")
MouseAction.escapes = MouseAction("mouse escapes")
MouseAction.trapped = MouseAction("mouse trapped")
MouseAction.removed = MouseAction("mouse removed")
```

You'll note that **__cmp__( )** has been overidden to implement a comparison between **action** values. Also, each possible move by a mouse is enumerated as a **MouseAction** object, all of which are static fields in **MouseAction**.

For creating test code, a sequence of mouse inputs is provided from a text file:

[1] No mice were harmed in the creation of this example.

```
# StateMachine/mouse/MouseMoves.txt
mouse appears
mouse runs away
mouse appears
mouse enters trap
mouse escapes
mouse appears
mouse enters trap
mouse trapped
mouse removed
mouse appears
mouse runs away
mouse appears
mouse enters trap
mouse trapped
mouse removed
```

With these tools in place, it's now possible to create the first version of the mousetrap program. Each **State** subclass defines its **run( )** behavior, and also establishes its next state with an **if-else** clause:

```
# StateMachine/mousetrap1/MouseTrapTest.py
# State Machine pattern using 'if' statements
# to determine the next state.
import string, sys
sys.path += ['../stateMachine', '../mouse']
from State import State
from StateMachine import StateMachine
from MouseAction import MouseAction
# A different subclass for each state:

class Waiting(State):
    def run(self):
        print("Waiting: Broadcasting cheese smell")

    def next(self, input):
        if input == MouseAction.appears:
            return MouseTrap.luring
        return MouseTrap.waiting

class Luring(State):
    def run(self):
        print("Luring: Presenting Cheese, door open")

    def next(self, input):
        if input == MouseAction.runsAway:
            return MouseTrap.waiting
        if input == MouseAction.enters:
            return MouseTrap.trapping
        return MouseTrap.luring

class Trapping(State):
    def run(self):
        print("Trapping: Closing door")

    def next(self, input):
        if input == MouseAction.escapes:
            return MouseTrap.waiting
        if input == MouseAction.trapped:
```

```
            return MouseTrap.holding
        return MouseTrap.trapping

class Holding(State):
    def run(self):
        print("Holding: Mouse caught")

    def next(self, input):
        if input == MouseAction.removed:
            return MouseTrap.waiting
        return MouseTrap.holding

class MouseTrap(StateMachine):
    def __init__(self):
        # Initial state
        StateMachine.__init__(self, MouseTrap.waiting)

# Static variable initialization:
MouseTrap.waiting = Waiting()
MouseTrap.luring = Luring()
MouseTrap.trapping = Trapping()
MouseTrap.holding = Holding()

moves = map(string.strip,
  open("../mouse/MouseMoves.txt").readlines())
MouseTrap().runAll(map(MouseAction, moves))
```

The **StateMachine** class simply defines all the possible states as static objects, and also sets up the initial state. The **UnitTest** creates a **MouseTrap** and then tests it with all the inputs from a **MouseMoveList**.

While the use of **if** statements inside the **next( )** methods is perfectly reasonable, managing a large number of these could become difficult. Another approach is to create tables inside each **State** object defining the various next states based on the input.

Initially, this seems like it ought to be quite simple. You should be able to define a static table in each **State** subclass that defines the transitions in terms of the other **State** objects. However, it turns out that this approach generates cyclic initialization dependencies. To solve the problem, I've had to delay the initialization of the tables until the first time that the **next( )** method is called for a particular **State** object. Initially, the **next( )** methods can appear a little strange because of this.

The **StateT** class is an implementation of **State** (so that the same **StateMachine** class can be used from the previous example) that adds a **Map** and a method to initialize the map from a two-dimensional array. The **next( )** method has a base-class implementation which must be called from the overridden derived class **next( )** methods after they test for a **null Map** (and initialize it if it's **null**):

```
# StateMachine/mousetrap2/MouseTrap2Test.py
# A better mousetrap using tables
import string, sys
sys.path += ['../stateMachine', '../mouse']
from State import State
from StateMachine import StateMachine
from MouseAction import MouseAction

class StateT(State):
    def __init__(self):
        self.transitions = None
    def next(self, input):
        if self.transitions.has_key(input):
```

```
            return self.transitions[input]
        else:
            raise "Input not supported for current state"

class Waiting(StateT):
    def run(self):
        print("Waiting: Broadcasting cheese smell")
    def next(self, input):
        # Lazy initialization:
        if not self.transitions:
            self.transitions = {
              MouseAction.appears : MouseTrap.luring
            }
        return StateT.next(self, input)

class Luring(StateT):
    def run(self):
        print("Luring: Presenting Cheese, door open")
    def next(self, input):
        # Lazy initialization:
        if not self.transitions:
            self.transitions = {
              MouseAction.enters : MouseTrap.trapping,
              MouseAction.runsAway : MouseTrap.waiting
            }
        return StateT.next(self, input)

class Trapping(StateT):
    def run(self):
        print("Trapping: Closing door")
    def next(self, input):
        # Lazy initialization:
        if not self.transitions:
            self.transitions = {
              MouseAction.escapes : MouseTrap.waiting,
              MouseAction.trapped : MouseTrap.holding
            }
        return StateT.next(self, input)

class Holding(StateT):
    def run(self):
        print("Holding: Mouse caught")
    def next(self, input):
        # Lazy initialization:
        if not self.transitions:
            self.transitions = {
              MouseAction.removed : MouseTrap.waiting
            }
        return StateT.next(self, input)

class MouseTrap(StateMachine):
    def __init__(self):
        # Initial state
        StateMachine.__init__(self, MouseTrap.waiting)

# Static variable initialization:
MouseTrap.waiting = Waiting()
MouseTrap.luring = Luring()
```

```
MouseTrap.trapping = Trapping()
MouseTrap.holding = Holding()

moves = map(string.strip,
  open("../mouse/MouseMoves.txt").readlines())
mouseMoves = map(MouseAction, moves)
MouseTrap().runAll(mouseMoves)
```

The rest of the code is identical - the difference is in the **next( )** methods and the **StateT** class.

If you have to create and maintain a lot of **State** classes, this approach is an improvement, since it's easier to quickly read and understand the state transitions from looking at the table.

## Table-Driven State Machine

The advantage of the previous design is that all the information about a state, including the state transition information, is located within the state class itself. This is generally a good design principle.

However, in a pure state machine, the machine can be completely represented by a single state-transition table. This has the advantage of locating all the information about the state machine in a single place, which means that you can more easily create and maintain the table based on a classic state-transition diagram.

The classic state-transition diagram uses a circle to represent each state, and lines from the state pointing to all states that state can transition into. Each transition line is annotated with conditions for transition and an action during transition. Here's what it looks like:

(Simple State Machine Diagram)

Goals:

- Direct translation of state diagram
- Vector of change: the state diagram representation
- Reasonable implementation
- No excess of states (you could represent every single change with a new state)
- Simplicity and flexibility

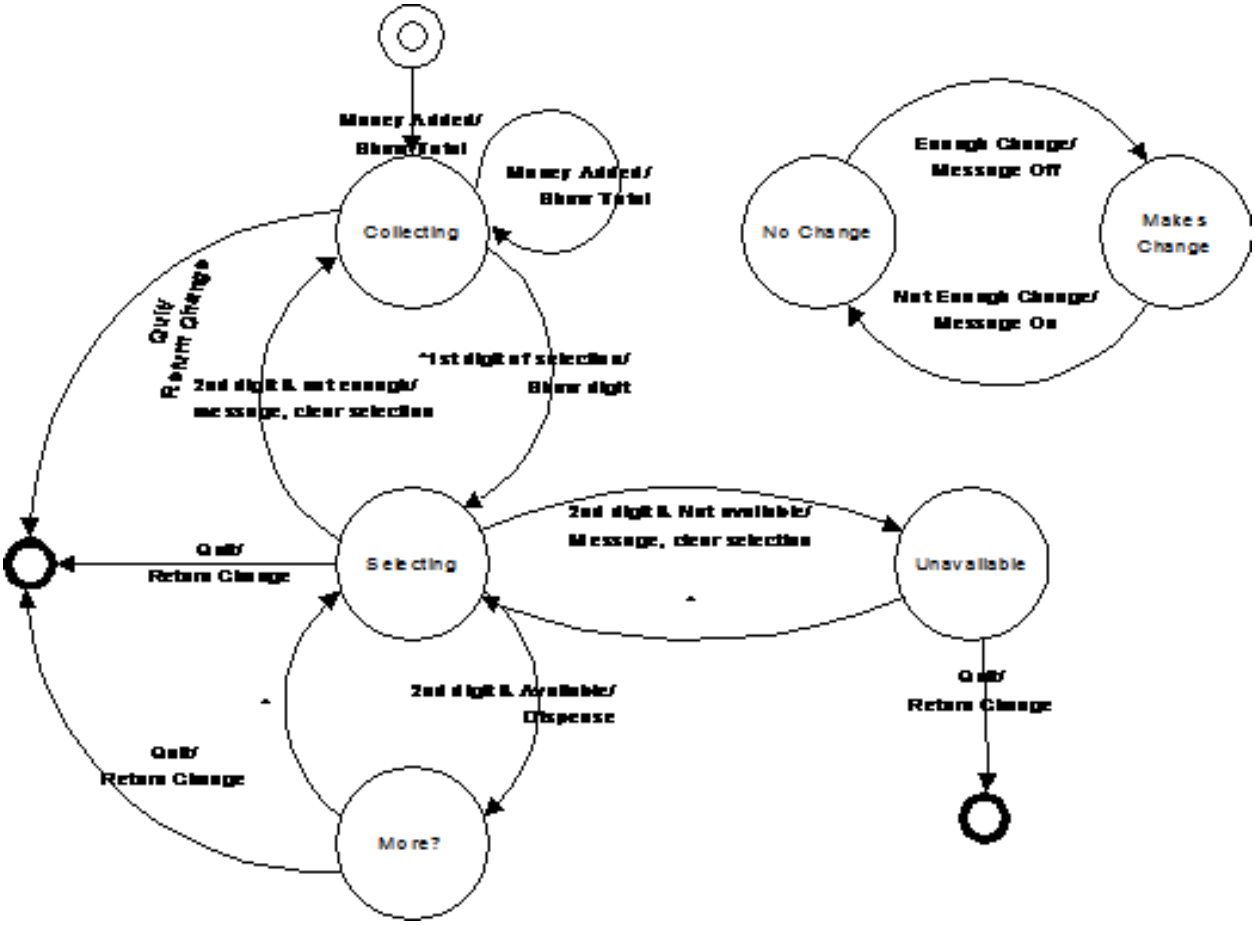
Observations:

- States are trivial - no information or functions/data, just an identity
- Not like the State pattern!
- The machine governs the move from state to state
- Similar to flyweight
- Each state may move to many others
- Condition & action functions must also be external to states
- Centralize description in a single table containing all variations, for ease of configuration

Example:

- State Machine & Table-Driven Code
- Implements a vending machine
- Uses several other patterns

- Separates common state-machine code from specific application (like template method)
- Each input causes a seek for appropriate solution (like chain of responsibility)
- Tests and transitions are encapsulated in function objects (objects that hold functions)
- Java constraint: methods are not first-class objects




## The State Class

The **State** class is distinctly different from before, since it is really just a placeholder with a name. Thus it is not inherited from previous **State** classes:

```
# StateMachine/stateMachine2/State.py

class State:
    def __init__(self, name): self.name = name
    def __str__(self): return self.name
```

## Conditions for Transition

In the state transition diagram, an input is tested to see if it meets the condition necessary to transfer to the state under question. As before, the **Input** is just a tagging interface:

```
# StateMachine/stateMachine2/Input.py
# Inputs to a state machine

class Input: pass
```

The **Condition** evaluates the **Input** to decide whether this row in the table is the correct transition:

```
# StateMachine/stateMachine2/Condition.py
# Condition function object for state machine

class Condition:
    boolean condition(input) :
        assert 0, "condition() not implemented"
```

## Transition Actions

If the **Condition** returns **true**, then the transition to a new state is made, and as that transition is made some kind of action occurs (in the previous state machine design, this was the **run( )** method):

```
# StateMachine/stateMachine2/Transition.py
# Transition function object for state machine

class Transition:
    def transition(self, input):
        assert 0, "transition() not implemented"
```

## The Table

With these classes in place, we can set up a 3-dimensional table where each row completely describes a state. The first element in the row is the current state, and the rest of the elements are each a row indicating what the *type* of the input can be, the condition that must be satisfied in order for this state change to be the correct one, the action that happens during transition, and the new state to move into. Note that the **Input** object is not just used for its type, it is also a *Messenger* object that carries information to the **Condition** and **Transition** objects:

```
{(CurrentState, InputA) : (ConditionA, TransitionA, NextA),
 (CurrentState, InputB) : (ConditionB, TransitionB, NextB),
 (CurrentState, InputC) : (ConditionC, TransitionC, NextC),
 ...
}
```

## The Basic Machine

Here's the basic machine, (code only roughly converted):

```
# StateMachine/stateMachine2/StateMachine.py
# A table-driven state machine

class StateMachine:
    def __init__(self, initialState, tranTable):
        self.state = initialState
        self.transitionTable = tranTable
```

```
    def nextState(self, input):

        Iterator it=((List)map.get(state)).iterator()
        while(it.hasNext()):
            Object[] tran = (Object[])it.next()
            if(input == tran[0] ||
               input.getClass() == tran[0]):
                if(tran[1] != null):
                    Condition c = (Condition)tran[1]
                    if(!c.condition(input))
                        continue # Failed test

                if(tran[2] != null)
                    ((Transition)tran[2]).transition(input)
                state = (State)tran[3]
                return


        throw RuntimeException(
          "Input not supported for current state")
```

## Simple Vending Machine

Here's the simple vending machine, (code only roughly converted):

```
# StateMachine/vendingmachine/VendingMachine.py
# Demonstrates use of StateMachine.py
import sys
sys.path += ['../stateMachine2']
import StateMachine

class State:
    def __init__(self, name): self.name = name
    def __str__(self): return self.name

State.quiescent = State("Quiesecent")
State.collecting = State("Collecting")
State.selecting = State("Selecting")
State.unavailable = State("Unavailable")
State.wantMore = State("Want More?")
State.noChange = State("Use Exact Change Only")
State.makesChange = State("Machine makes change")

class HasChange:
    def __init__(self, name): self.name = name
    def __str__(self): return self.name

HasChange.yes = HasChange("Has change")
HasChange.no = HasChange("Cannot make change")

class ChangeAvailable(StateMachine):
    def __init__(self):
        StateMachine.__init__(State.makesChange, {
          # Current state, input
          (State.makesChange, HasChange.no) :
```

```
            # test, transition, next state:
            (null, null, State.noChange),
          (State.noChange, HasChange.yes) :
            (null, null, State.noChange)
        })

class Money:
    def __init__(self, name, value):
        self.name = name
        self.value = value
    def __str__(self): return self.name
    def getValue(self): return self.value

Money.quarter = Money("Quarter", 25)
Money.dollar = Money("Dollar", 100)

class Quit:
    def __str__(self): return "Quit"

Quit.quit = Quit()

class Digit:
    def __init__(self, name, value):
        self.name = name
        self.value = value
    def __str__(self): return self.name
    def getValue(self): return self.value

class FirstDigit(Digit): pass
FirstDigit.A = FirstDigit("A", 0)
FirstDigit.B = FirstDigit("B", 1)
FirstDigit.C = FirstDigit("C", 2)
FirstDigit.D = FirstDigit("D", 3)

class SecondDigit(Digit): pass
SecondDigit.one = SecondDigit("one", 0)
SecondDigit.two = SecondDigit("two", 1)
SecondDigit.three = SecondDigit("three", 2)
SecondDigit.four = SecondDigit("four", 3)

class ItemSlot:
    id = 0
    def __init__(self, price, quantity):
        self.price = price
        self.quantity = quantity
    def __str__(self): return `ItemSlot.id`
    def getPrice(self): return self.price
    def getQuantity(self): return self.quantity
    def decrQuantity(self): self.quantity -= 1

class VendingMachine(StateMachine):
    changeAvailable = ChangeAvailable()
    amount = 0
    FirstDigit first = null
    ItemSlot[][] items = ItemSlot[4][4]

    # Conditions:
    def notEnough(self, input):
```

```
        i1 = first.getValue()
        i2 = input.getValue()
        return items[i1][i2].getPrice() > amount

    def itemAvailable(self, input):
        i1 = first.getValue()
        i2 = input.getValue()
        return items[i1][i2].getQuantity() > 0

    def itemNotAvailable(self, input):
        return !itemAvailable.condition(input)
        #i1 = first.getValue()
        #i2 = input.getValue()
        #return items[i1][i2].getQuantity() == 0

    # Transitions:
    def clearSelection(self, input):
        i1 = first.getValue()
        i2 = input.getValue()
        ItemSlot is = items[i1][i2]
        print (
          "Clearing selection: item " + is +
          " costs " + is.getPrice() +
          " and has quantity " + is.getQuantity())
        first = null

    def dispense(self, input):
        i1 = first.getValue()
        i2 = input.getValue()
        ItemSlot is = items[i1][i2]
        print(("Dispensing item " +
          is + " costs " + is.getPrice() +
          " and has quantity " + is.getQuantity()))
        items[i1][i2].decrQuantity()
        print ("Quantity " +
          is.getQuantity())
        amount -= is.getPrice()
        print("Amount remaining " +
          amount)

    def showTotal(self, input):
        amount += ((Money)input).getValue()
        print("Total amount = " + amount)

    def returnChange(self, input):
        print("Returning " + amount)
        amount = 0

    def showDigit(self, input):
        first = (FirstDigit)input
        print("First Digit= "+ first)

    def __init__(self):
        StateMachine.__init__(self, State.quiescent)
        for(int i = 0 i < items.length i++)
            for(int j = 0 j < items[i].length j++)
                items[i][j] = ItemSlot((j+1)*25, 5)
        items[3][0] = ItemSlot(25, 0)
```

```
        """
        buildTable(Object[][][]{
         ::State.quiescent, # Current state
            # Input, test, transition, next state:
           :Money.class, null,
             showTotal, State.collecting,
         ::State.collecting, # Current state
            # Input, test, transition, next state:
           :Quit.quit, null,
             returnChange, State.quiescent,
           :Money.class, null,
             showTotal, State.collecting,
           :FirstDigit.class, null,
             showDigit, State.selecting,
         ::State.selecting, # Current state
            # Input, test, transition, next state:
           :Quit.quit, null,
             returnChange, State.quiescent,
           :SecondDigit.class, notEnough,
             clearSelection, State.collecting,
           :SecondDigit.class, itemNotAvailable,
             clearSelection, State.unavailable,
           :SecondDigit.class, itemAvailable,
             dispense, State.wantMore,
         ::State.unavailable, # Current state
            # Input, test, transition, next state:
           :Quit.quit, null,
             returnChange, State.quiescent,
           :FirstDigit.class, null,
             showDigit, State.selecting,
         ::State.wantMore, # Current state
            # Input, test, transition, next state:
           :Quit.quit, null,
             returnChange, State.quiescent,
           :FirstDigit.class, null,
             showDigit, State.selecting,
        )
        """
```

## Testing the Machine

Here's a test of the machine, (code only roughly converted):

```
# StateMachine/vendingmachine/VendingMachineTest.py
# Demonstrates use of StateMachine.py

vm = VendingMachine()
for input in [
    Money.quarter,
    Money.quarter,
    Money.dollar,
    FirstDigit.A,
    SecondDigit.two,
    FirstDigit.A,
    SecondDigit.two,
    FirstDigit.C,
```

```
    SecondDigit.three,
    FirstDigit.D,
    SecondDigit.one,
    Quit.quit]:
    vm.nextState(input)
```

## Tools

Another approach, as your state machine gets bigger, is to use an automation tool whereby you configure a table and let the tool generate the state machine code for you. This can be created yourself using a language like Python, but there are also free, open-source tools such as *Libero*, at http://www.imatix.com.

## Exercises

1. Create an example of the "virtual proxy."
2. Create an example of the "Smart reference" proxy where you keep count of the number of method calls to a particular object.
3. Create a program similar to certain DBMS systems that only allow a certain number of connections at any time. To implement this, use a singleton-like system that controls the number of "connection" objects that it creates. When a user is finished with a connection, the system must be informed so that it can check that connection back in to be reused. To guarantee this, provide a proxy object instead of a reference to the actual connection, and design the proxy so that it will cause the connection to be released back to the system.
4. Using the *State*, make a class called **UnpredictablePerson** which changes the kind of response to its **hello( )** method depending on what kind of **Mood** it's in. Add an additional kind of **Mood** called **Prozac**.
5. Create a simple copy-on write implementation.
6. Apply **TransitionTable.py** to the "Washer" problem.
7. Create a *StateMachine* system whereby the current state along with input information determines the next state that the system will be in. To do this, each state must store a reference back to the proxy object (the state controller) so that it can request the state change. Use a **HashMap** to create a table of states, where the key is a **String** naming the new state and the value is the new state object. Inside each state subclass override a method **nextState( )** that has its own state-transition table. The input to **nextState( )** should be a single word that comes from a text file containing one word per line.
8. Modify the previous exercise so that the state machine can be configured by creating/modifying a single multi-dimensional array.
9. Modify the "mood" exercise from the previous session so that it becomes a state machine using StateMachine.py
10. Create an elevator state machine system using StateMachine.py
11. Create a heating/air-conditioning system using StateMachine.py
12. A *generator* is an object that produces other objects, just like a factory, except that the generator function doesn't require any arguments. Create a **MouseMoveGenerator** which produces correct **MouseMove** actions as outputs each time the generator function is called (that is, the mouse must move in the proper sequence, thus the possible moves are based on the previous move - it's another state

machine). Add a method to produce an iterator, but this method should take an **int** argument that specifies the number of moves to produce before **hasNext()** returns **false**.

# CHAPTER 30

## Decorator: Dynamic Type Selection

**Note:** I think we can rewrite this chapter to use Python decorators as implementation (thus the decorators chapter should precede this one).

The use of layered objects to dynamically and transparently add responsibilities to individual objects is referred to as the *decorator* pattern.

Used when subclassing creates too many (& inflexible) classes

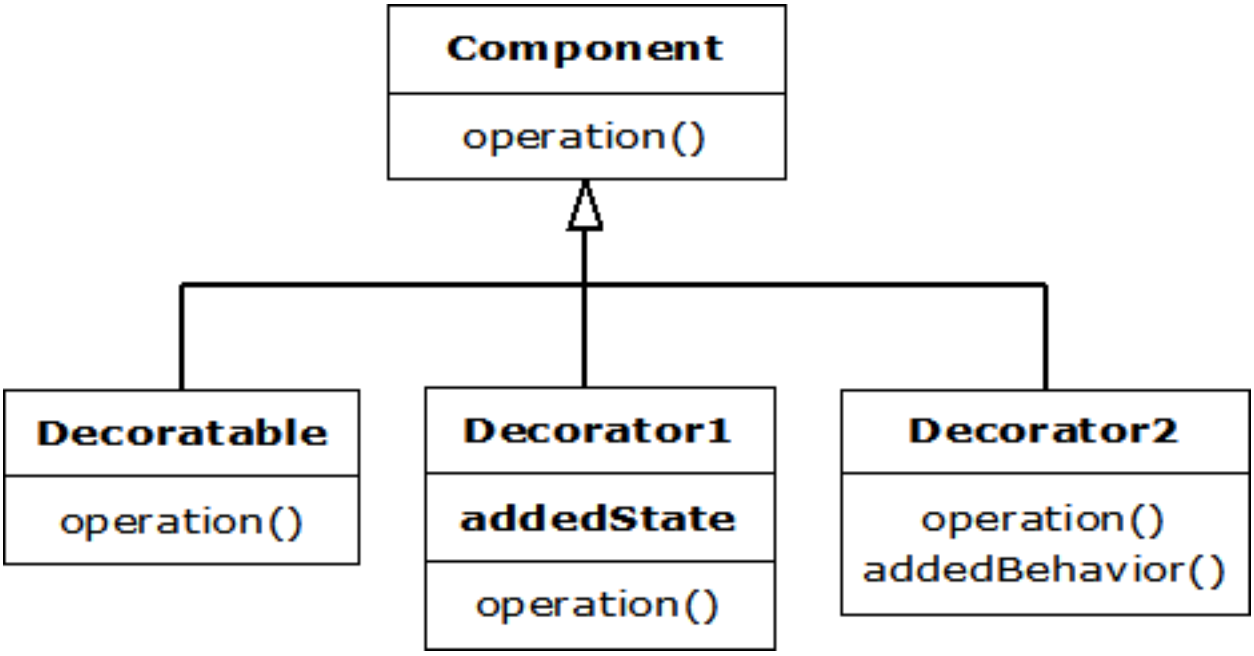
All decorators that wrap around the original object must have the same basic interface

Dynamic proxy/surrogate?

This accounts for the odd inheritance structure

Tradeoff: coding is more complicated when using decorators

## Basic Decorator Structure




## A Coffee Example

Consider going down to the local coffee shop, *BeanMeUp*, for a coffee. There are typically many different drinks on offer – espressos, lattes, teas, iced coffees, hot chocolate to name a few, as well as a number of extras (which cost extra too) such as whipped cream or an extra shot of espresso. You can also make certain changes to your drink at no extra cost, such as asking for decaf coffee instead of regular coffee.

Quite clearly if we are going to model all these drinks and combinations, there will be sizeable class diagrams. So for clarity we will only consider a subset of the coffees: Espresso, Espresso Con Panna, Café Late, Cappuccino and Café Mocha. We'll include 2 extras - whipped cream ("whipped") and an extra shot of espresso; and three changes - decaf, steamed milk ("wet") and foamed milk ("dry").

## Class for Each Combination

One solution is to create an individual class for every combination. Each class describes the drink and is responsible for the cost etc. The resulting menu is huge, and a part of the class diagram would look something like this:


The key to using this method is to find the particular combination you want. So, once you've found the drink you would like, here is how you would use it, as shown in the **CoffeeShop** class in the following code:

```
# Decorator/nodecorators/CoffeeShop.py
# Coffee example with no decorators

class Espresso: pass
class DoubleEspresso: pass
class EspressoConPanna: pass

class Cappuccino:
    def __init__(self):
        self.cost = 1
        self.description = "Cappucino"
    def getCost(self):
        return self.cost
    def getDescription(self):
        return self.description

class CappuccinoDecaf: pass
class CappuccinoDecafWhipped: pass
class CappuccinoDry: pass
class CappuccinoDryWhipped: pass
class CappuccinoExtraEspresso: pass
class CappuccinoExtraEspressoWhipped: pass
class CappuccinoWhipped: pass

class CafeMocha: pass
class CafeMochaDecaf: pass
class CafeMochaDecafWhipped:
    def __init__(self):
        self.cost = 1.25
        self.description = \
          "Cafe Mocha decaf whipped cream"
    def getCost(self):
```

```
        return self.cost
    def getDescription(self):
        return self.description

class CafeMochaExtraEspresso: pass
class CafeMochaExtraEspressoWhipped: pass
class CafeMochaWet: pass
class CafeMochaWetWhipped: pass
class CafeMochaWhipped: pass

class CafeLatte: pass
class CafeLatteDecaf: pass
class CafeLatteDecafWhipped: pass
class CafeLatteExtraEspresso: pass
class CafeLatteExtraEspressoWhipped: pass
class CafeLatteWet: pass
class CafeLatteWetWhipped: pass
class CafeLatteWhipped: pass

cappuccino = Cappuccino()
print((cappuccino.getDescription() + ": $" +
  `cappuccino.getCost()`))

cafeMocha = CafeMochaDecafWhipped()
print((cafeMocha.getDescription()
  + ": $" + `cafeMocha.getCost()`))
```

And here is the corresponding output:

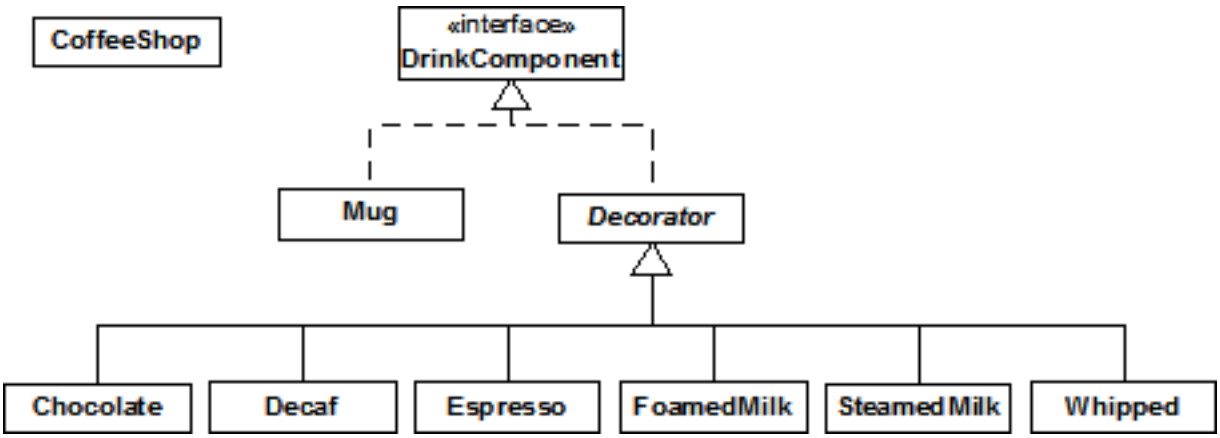
```
Cappucino: $1.0
Cafe Mocha decaf whipped cream: $1.25
```

You can see that creating the particular combination you want is easy, since you are just creating an instance of a class. However, there are a number of problems with this approach. Firstly, the combinations are fixed statically so that any combination a customer may wish to order needs to be created up front. Secondly, the resulting menu is so huge that finding your particular combination is difficult and time consuming.

## The Decorator Approach

Another approach would be to break the drinks down into the various components such as espresso and foamed milk, and then let the customer combine the components to describe a particular coffee.

In order to do this programmatically, we use the Decorator pattern. A Decorator adds responsibility to a component by wrapping it, but the Decorator conforms to the interface of the component it encloses, so the wrapping is transparent. Decorators can also be nested without the loss of this transparency.




Methods invoked on the Decorator can in turn invoke methods in the component, and can of course perform processing before or after the invocation.

So if we added **getTotalCost()** and **getDescription()** methods to the **DrinkComponent** interface, an Espresso looks like this:

```
# Decorator/alldecorators/EspressoDecorator.py

class Espresso(Decorator):
    cost = 0.75f
    description = " espresso"
    def __init__(DrinkComponent):
        Decorator.__init__(self, component)

    def getTotalCost(self):
        return self.component.getTotalCost() + cost

    def getDescription(self):
        return self.component.getDescription() +
            description
```

You combine the components to create a drink as follows, as shown in the code below:

```
# Decorator/alldecorators/CoffeeShop.py
# Coffee example using decorators

class DrinkComponent:
    def getDescription(self):
        return self.__class__.__name__
    def getTotalCost(self):
        return self.__class__.cost

class Mug(DrinkComponent):
    cost = 0.0

class Decorator(DrinkComponent):
    def __init__(self, drinkComponent):
        self.component = drinkComponent
    def getTotalCost(self):
        return self.component.getTotalCost() + \
          DrinkComponent.getTotalCost(self)
    def getDescription(self):
        return self.component.getDescription() + \
          ' ' + DrinkComponent.getDescription(self)
```

```
class Espresso(Decorator):
    cost = 0.75
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Decaf(Decorator):
    cost = 0.0
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class FoamedMilk(Decorator):
    cost = 0.25
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class SteamedMilk(Decorator):
    cost = 0.25
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Whipped(Decorator):
    cost = 0.25
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Chocolate(Decorator):
    cost = 0.25
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

cappuccino = Espresso(FoamedMilk(Mug()))
print(cappuccino.getDescription().strip() + \)
  ": $" + `cappuccino.getTotalCost()`

cafeMocha = Espresso(SteamedMilk(Chocolate(
  Whipped(Decaf(Mug())))))

print(cafeMocha.getDescription().strip() + \)
  ": $" + `cafeMocha.getTotalCost()`
```

This approach would certainly provide the most flexibility and the smallest menu. You have a small number of components to choose from, but assembling the description of the coffee then becomes rather arduous.

If you want to describe a plain cappuccino, you create it with:

```
plainCap = Espresso(FoamedMilk(Mug()))
```

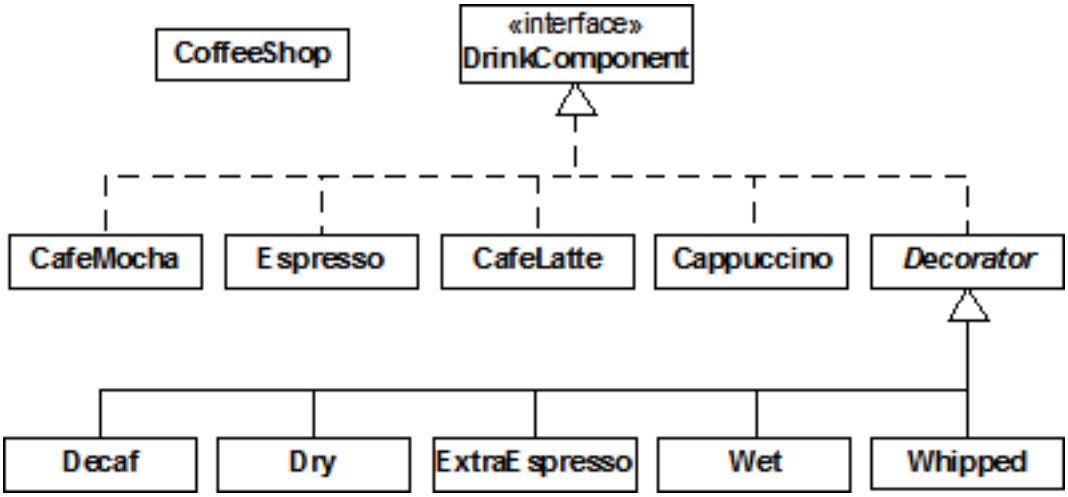
Creating a decaf Cafe Mocha with whipped cream requires an even longer description.

## Compromise

The previous approach takes too long to describe a coffee. There will also be certain combinations that you will describe regularly, and it would be convenient to have a quick way of describing them.

The 3rd approach is a mixture of the first 2 approaches, and combines flexibility with ease of use. This

compromise is achieved by creating a reasonably sized menu of basic selections, which would often work exactly as they are, but if you wanted to decorate them (whipped cream, decaf etc.) then you would use decorators to make the modifications. This is the type of menu you are presented with in most coffee shops.




Here is how to create a basic selection, as well as a decorated selection:

```
# Decorator/compromise/CoffeeShop.py
# Coffee example with a compromise of basic
# combinations and decorators

class DrinkComponent:
    def getDescription(self):
        return self.__class__.__name__
    def getTotalCost(self):
        return self.__class__.cost

class Espresso(DrinkComponent):
    cost = 0.75

class EspressoConPanna(DrinkComponent):
    cost = 1.0

class Cappuccino(DrinkComponent):
    cost = 1.0

class CafeLatte(DrinkComponent):
    cost = 1.0

class CafeMocha(DrinkComponent):
    cost = 1.25

class Decorator(DrinkComponent):
    def __init__(self, drinkComponent):
        self.component = drinkComponent
    def getTotalCost(self):
        return self.component.getTotalCost() + \
          DrinkComponent.getTotalCost(self)
    def getDescription(self):
        return self.component.getDescription() + \
          ' ' + DrinkComponent.getDescription(self)
```

```
class ExtraEspresso(Decorator):
    cost = 0.75
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Whipped(Decorator):
    cost = 0.50
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Decaf(Decorator):
    cost = 0.0
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Dry(Decorator):
    cost = 0.0
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Wet(Decorator):
    cost = 0.0
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

cappuccino = Cappuccino()
print(cappuccino.getDescription() + ": $" + \)
  `cappuccino.getTotalCost()`

cafeMocha = Whipped(Decaf(CafeMocha()))
print(cafeMocha.getDescription() + ": $" + \)
  `cafeMocha.getTotalCost()`
```

You can see that creating a basic selection is quick and easy, which makes sense since they will be described regularly. Describing a decorated drink is more work than when using a class per combination, but clearly less work than when only using decorators.

The final result is not too many classes, but not too many decorators either. Most of the time it's possible to get away without using any decorators at all, so we have the benefits of both approaches.

## Other Considerations

What happens if we decide to change the menu at a later stage, such as by adding a new type of drink? If we had used the class per combination approach, the effect of adding an extra such as syrup would be an exponential growth in the number of classes. However, the implications to the all decorator or compromise approaches are the same - one extra class is created.

How about the effect of changing the cost of steamed milk and foamed milk, when the price of milk goes up? Having a class for each combination means that you need to change a method in each class, and thus maintain many classes. By using decorators, maintenance is reduced by defining the logic in one place.

## Further Reading

## Exercises

1. Add a Syrup class to the decorator approach described above. Then create a Café Latte (you'll need to use steamed milk with an espresso) with syrup.
2. Repeat Exercise 1 for the compromise approach.
3. Implement the decorator pattern to create a Pizza restaurant, which has a set menu of choices as well as the option to design your own pizza. Follow the compromise approach to create a menu consisting of a Margherita, Hawaiian, Regina, and Vegetarian pizzas, with toppings (decorators) of Garlic, Olives, Spinach, Avocado, Feta and Pepperdews. Create a Hawaiian pizza, as well as a Margherita decorated with Spinach, Feta, Pepperdews and Olives.

CHAPTER 31

# Iterators: Decoupling Algorithms from Containers

**Note:** This chapter has not had any significant translation yet.

Alexander Stepanov thought for years about the problem of generic programming techniques before creating the STL (along with Dave Musser). He came to the conclusion that all algorithms are defined on algebraic structures - what we would call containers.

In the process, he realized that iterators are central to the use of algorithms, because they decouple the algorithms from the specific type of container that the algorithm might currently be working with. This means that you can describe the algorithm without worrying about the particular sequence it is operating on. More generally, *any* code that you write using iterators is decoupled from the data structure that the code is manipulating, and thus your code is more general and reusable.

The use of iterators also extends your code into the realm of *functional programming*, whose objective is to describe *what* a program is doing at every step rather than *how* it is doing it. That is, you say "sort" rather than describing the sort. The objective of the C++ STL was to provide this *generic programming* approach for C++ (how successful this approach will actually be remains to be seen).

If you've used containers in Java (and it's hard to write code without using them), you've used iterators - in the form of the **Enumeration** in Java 1.0/1.1 and the **Iterator** in Java 2. So you should already be familiar with their general use. If not, see Chapter 9, *Holding Your Objects*, under *Iterators* in *Thinking in Java, 3rd edition* (freely downloadable from *www.BruceEckel.com*).

Because the Java 2 containers rely heavily on iterators they become excellent candidates for generic/functional programming techniques. This chapter will explore these techniques by converting the STL algorithms to Java, for use with the Java 2 container library.

## Type-Safe Iterators

In *Thinking in Java*, I show the creation of a type-safe container that will only accept a particular type of object. A reader, Linda Pazzaglia, asked for the other obvious type-safe component, an iterator that would

work with the basic **java.util** containers, but impose the constraint that the type of objects that it iterates over be of a particular type.

If Java ever includes a template mechanism, this kind of iterator will have the added advantage of being able to return a specific type of object, but without templates you are forced to return generic **Object**s, or to require a bit of hand-coding for every type that you want to iterate through. I will take the former approach.

A second design decision involves the time that the type of object is determined. One approach is to take the type of the first object that the iterator encounters, but this is problematic because the containers may rearrange the objects according to an internal ordering mechanism (such as a hash table) and thus you may get different results from one iteration to the next. The safe approach is to require the user to establish the type during construction of the iterator.

Lastly, how do we build the iterator? We cannot rewrite the existing Java library classes that already produce **Enumeration**s and **Iterator**s. However, we can use the *Decorator* design pattern, and create a class that simply wraps the **Enumeration** or **Iterator** that is produced, generating a new object that has the iteration behavior that we want (which is, in this case, to throw a **RuntimeException** if an incorrect type is encountered) but with the same interface as the original **Enumeration** or **Iterator**, so that it can be used in the same places (you may argue that this is actually a *Proxy* pattern, but it's more likely *Decorator* because of its intent). Here is the code:

```
# Util/TypedIterator.py

class TypedIterator(Iterator):
    def __init__(self, it, type):
        self.imp = it
        self.type = type

    def hasNext(self):
        return imp.hasNext()

    def remove(self): imp.remove()
    def next(self):
        obj = imp.next()
        if(!type.isInstance(obj))
            throw ClassCastException(
              "TypedIterator for type " + type +
              " encountered type: " + obj.getClass())
        return obj
```

CHAPTER 32

# Factory: Encapsulating Object Creation

When you discover that you need to add new types to a system, the most sensible first step is to use polymorphism to create a common interface to those new types. This separates the rest of the code in your system from the knowledge of the specific types that you are adding. New types may be added without disturbing existing code ... or so it seems. At first it would appear that the only place you need to change the code in such a design is the place where you inherit a new type, but this is not quite true. You must still create an object of your new type, and at the point of creation you must specify the exact constructor to use. Thus, if the code that creates objects is distributed throughout your application, you have the same problem when adding new types-you must still chase down all the points of your code where type matters. It happens to be the *creation* of the type that matters in this case rather than the *use* of the type (which is taken care of by polymorphism), but the effect is the same: adding a new type can cause problems.

The solution is to force the creation of objects to occur through a common *factory* rather than to allow the creational code to be spread throughout your system. If all the code in your program must go through this factory whenever it needs to create one of your objects, then all you must do when you add a new object is to modify the factory.

Since every object-oriented program creates objects, and since it's very likely you will extend your program by adding new types, I suspect that factories may be the most universally useful kinds of design patterns.

## Simple Factory Method

As an example, let's revisit the **Shape** system.

One approach is to make the factory a **static** method of the base class:

```
# Factory/shapefact1/ShapeFactory1.py
# A simple static factory method.
from __future__ import generators
import random

class Shape(object):
    # Create based on class name:
    def factory(type):
```

```
        #return eval(type + "()")
        if type == "Circle": return Circle()
        if type == "Square": return Square()
        assert 0, "Bad shape creation: " + type
    factory = staticmethod(factory)

class Circle(Shape):
    def draw(self): print("Circle.draw")
    def erase(self): print("Circle.erase")

class Square(Shape):
    def draw(self): print("Square.draw")
    def erase(self): print("Square.erase")

# Generate shape name strings:
def shapeNameGen(n):
    types = Shape.__subclasses__()
    for i in range(n):
        yield random.choice(types).__name__

shapes = \
  [ Shape.factory(i) for i in shapeNameGen(7)]

for shape in shapes:
    shape.draw()
    shape.erase()
```

The **factory( )** takes an argument that allows it to determine what type of **Shape** to create; it happens to be a **String** in this case but it could be any set of data. The **factory( )** is now the only other code in the system that needs to be changed when a new type of **Shape** is added (the initialization data for the objects will presumably come from somewhere outside the system, and not be a hard-coded array as in the above example).

Note that this example also shows the new Python 2.2 **staticmethod( )** technique for creating static methods in a class.

I have also used a tool which is new in Python 2.2 called a *generator*. A generator is a special case of a factory: it's a factory that takes no arguments in order to create a new object. Normally you hand some information to a factory in order to tell it what kind of object to create and how to create it, but a generator has some kind of internal algorithm that tells it what and how to build. It "generates out of thin air" rather than being told what to create.

Now, this may not look consistent with the code you see above:

```
for i in shapeNameGen(7)
```

looks like there's an initialization taking place. This is where a generator is a bit strange - when you call a function that contains a **yield** statement (**yield** is a new keyword that determines that a function is a generator), that function actually returns a generator object that has an iterator. This iterator is implicitly used in the **for** statement above, so it appears that you are iterating through the generator function, not what it returns. This was done for convenience of use.

Thus, the code that you write is actually a kind of factory, that creates the generator objects that do the actual generation. You can use the generator explicitly if you want, for example:

```
gen = shapeNameGen(7)
print(gen.next())
```

So **next( )** is the iterator method that's actually called to generate the next object, and it takes no arguments. **shapeNameGen( )** is the factory, and **gen** is the generator.

Inside the generator-factory, you can see the call to **__subclasses__( )**, which produces a list of references to each of the subclasses of **Shape** (which must be inherited from **object** for this to work). You should be aware, however, that this only works for the first level of inheritance from **Item**, so if you were to inherit a new class from **Circle**, it wouldn't show up in the list generated by **__subclasses__( )**. If you need to create a deeper hierarchy this way, you must recurse the **__subclasses__( )** list.

Also note that in **shapeNameGen( )** the statement:

```
types = Shape.__subclasses__()
```

Is only executed when the generator object is produced; each time the **next( )** method of this generator object is called (which, as noted above, may happen implicitly), only the code in the **for** loop will be executed, so you don't have wasteful execution (as you would if this were an ordinary function).

### Preventing direct creation

To disallow direct access to the classes, you can nest the classes within the factory method, like this:

```
# Factory/shapefact1/NestedShapeFactory.py
import random

class Shape(object):
    types = []

def factory(type):
    class Circle(Shape):
        def draw(self): print("Circle.draw")
        def erase(self): print("Circle.erase")

    class Square(Shape):
        def draw(self): print("Square.draw")
        def erase(self): print("Square.erase")

    if type == "Circle": return Circle()
    if type == "Square": return Square()
    assert 0, "Bad shape creation: " + type

def shapeNameGen(n):
    for i in range(n):
        yield factory(random.choice(["Circle", "Square"]))

# Circle() # Not defined

for shape in shapeNameGen(7):
    shape.draw()
    shape.erase()
```

## Polymorphic Factories

The static **factory( )** method in the previous example forces all the creation operations to be focused in one spot, so that's the only place you need to change the code. This is certainly a reasonable solution, as it throws a box around the process of creating objects. However, the *Design Patterns* book emphasizes that

the reason for the *Factory Method* pattern is so that different types of factories can be subclassed from the basic factory (the above design is mentioned as a special case). However, the book does not provide an example, but instead just repeats the example used for the *Abstract Factory* (you'll see an example of this in the next section). Here is **ShapeFactory1.py** modified so the factory methods are in a separate class as virtual functions. Notice also that the specific **Shape** classes are dynamically loaded on demand:

```
# Factory/shapefact2/ShapeFactory2.py
# Polymorphic factory methods.
from __future__ import generators
import random

class ShapeFactory:
    factories = {}
    def addFactory(id, shapeFactory):
        ShapeFactory.factories.put[id] = shapeFactory
    addFactory = staticmethod(addFactory)
    # A Template Method:
    def createShape(id):
        if not ShapeFactory.factories.has_key(id):
            ShapeFactory.factories[id] = \
              eval(id + '.Factory()')
        return ShapeFactory.factories[id].create()
    createShape = staticmethod(createShape)

class Shape(object): pass

class Circle(Shape):
    def draw(self): print("Circle.draw")
    def erase(self): print("Circle.erase")
    class Factory:
        def create(self): return Circle()

class Square(Shape):
    def draw(self):
        print("Square.draw")
    def erase(self):
        print("Square.erase")
    class Factory:
        def create(self): return Square()

def shapeNameGen(n):
    types = Shape.__subclasses__()
    for i in range(n):
        yield random.choice(types).__name__

shapes = [ ShapeFactory.createShape(i)
           for i in shapeNameGen(7)]

for shape in shapes:
    shape.draw()
    shape.erase()
```

Now the factory method appears in its own class, **ShapeFactory**, as the **create( )** method. The different types of shapes must each create their own factory with a **create( )** method to create an object of their own type. The actual creation of shapes is performed by calling **ShapeFactory.createShape( )**, which is a static method that uses the dictionary in **ShapeFactory** to find the appropriate factory object based on an identifier that you pass it. The factory is immediately used to create the shape object, but you could imagine a more complex problem where the appropriate factory object is returned and then used by the

caller to create an object in a more sophisticated way. However, it seems that much of the time you don't need the intricacies of the polymorphic factory method, and a single static method in the base class (as shown in **ShapeFactory1.py**) will work fine.

Notice that the **ShapeFactory** must be initialized by loading its dictionary with factory objects, which takes place in the static initialization clause of each of the shape implementations.

## Abstract Factories

The *Abstract Factory* pattern looks like the factory objects we've seen previously, with not one but several factory methods. Each of the factory methods creates a different kind of object. The idea is that at the point of creation of the factory object, you decide how all the objects created by that factory will be used. The example given in *Design Patterns* implements portability across various graphical user interfaces (GUIs): you create a factory object appropriate to the GUI that you're working with, and from then on when you ask it for a menu, button, slider, etc. it will automatically create the appropriate version of that item for the GUI. Thus you're able to isolate, in one place, the effect of changing from one GUI to another.

As another example suppose you are creating a general-purpose gaming environment and you want to be able to support different types of games. Here's how it might look using an abstract factory:

```
# Factory/Games.py
# An example of the Abstract Factory pattern.

class Obstacle:
    def action(self): pass

class Character:
    def interactWith(self, obstacle): pass

class Kitty(Character):
    def interactWith(self, obstacle):
        print("Kitty has encountered a",
        obstacle.action())

class KungFuGuy(Character):
    def interactWith(self, obstacle):
        print("KungFuGuy now battles a",
        obstacle.action())

class Puzzle(Obstacle):
    def action(self):
        print("Puzzle")

class NastyWeapon(Obstacle):
    def action(self):
        print("NastyWeapon")

# The Abstract Factory:
class GameElementFactory:
    def makeCharacter(self): pass
    def makeObstacle(self): pass

# Concrete factories:
class KittiesAndPuzzles(GameElementFactory):
    def makeCharacter(self): return Kitty()
    def makeObstacle(self): return Puzzle()
```

```
class KillAndDismember(GameElementFactory):
    def makeCharacter(self): return KungFuGuy()
    def makeObstacle(self): return NastyWeapon()

class GameEnvironment:
    def __init__(self, factory):
        self.factory = factory
        self.p = factory.makeCharacter()
        self.ob = factory.makeObstacle()
    def play(self):
        self.p.interactWith(self.ob)

g1 = GameEnvironment(KittiesAndPuzzles())
g2 = GameEnvironment(KillAndDismember())
g1.play()
g2.play()
```

In this environment, **Character** objects interact with **Obstacle** objects, but there are different types of Characters and obstacles depending on what kind of game you're playing. You determine the kind of game by choosing a particular **GameElementFactory**, and then the **GameEnvironment** controls the setup and play of the game. In this example, the setup and play is very simple, but those activities (the *initial conditions* and the *state change*) can determine much of the game's outcome. Here, **GameEnvironment** is not designed to be inherited, although it could very possibly make sense to do that.

This also contains examples of *Double Dispatching* and the *Factory Method*, both of which will be explained later.

Of course, the above scaffolding of **Obstacle**, **Character** and **GameElementFactory** (which was translated from the Java version of this example) is unnecessary - it's only required for languages that have static type checking. As long as the concrete Python classes follow the form of the required classes, we don't need any base classes:

```
# Factory/Games2.py
# Simplified Abstract Factory.

class Kitty:
    def interactWith(self, obstacle):
        print("Kitty has encountered a",
        obstacle.action())

class KungFuGuy:
    def interactWith(self, obstacle):
        print("KungFuGuy now battles a",
        obstacle.action())

class Puzzle:
    def action(self): print("Puzzle")

class NastyWeapon:
    def action(self): print("NastyWeapon")

# Concrete factories:
class KittiesAndPuzzles:
    def makeCharacter(self): return Kitty()
    def makeObstacle(self): return Puzzle()

class KillAndDismember:
```

```
    def makeCharacter(self): return KungFuGuy()
    def makeObstacle(self): return NastyWeapon()

class GameEnvironment:
    def __init__(self, factory):
        self.factory = factory
        self.p = factory.makeCharacter()
        self.ob = factory.makeObstacle()
    def play(self):
        self.p.interactWith(self.ob)

g1 = GameEnvironment(KittiesAndPuzzles())
g2 = GameEnvironment(KillAndDismember())
g1.play()
g2.play()
```

Another way to put this is that all inheritance in Python is implementation inheritance; since Python does its type-checking at runtime, there's no need to use interface inheritance so that you can upcast to the base type.

You might want to study the two examples for comparison, however. Does the first one add enough useful information about the pattern that it's worth keeping some aspect of it? Perhaps all you need is "tagging classes" like this:

```
class Obstacle: pass
class Character: pass
class GameElementFactory: pass
```

Then the inheritance serves only to indicate the type of the derived classes.

## Exercises

1. Add a class **Triangle** to **ShapeFactory1.py**
2. Add a class **Triangle** to **ShapeFactory2.py**
3. Add a new type of **GameEnvironment** called **GnomesAndFairies** to **GameEnvironment.py**
4. Modify **ShapeFactory2.py** so that it uses an *Abstract Factory* to create different sets of shapes (for example, one particular type of factory object creates "thick shapes," another creates "thin shapes," but each factory object can create all the shapes: circles, squares, triangles etc.).

CHAPTER 33

# Function Objects

In *Advanced C++:Programming Styles And Idioms (Addison-Wesley, 1992)*, Jim Coplien coins the term *functor* which is an object whose sole purpose is to encapsulate a function (since "functor" has a meaning in mathematics, in this book I shall use the more explicit term *function object*). The point is to decouple the choice of function to be called from the site where that function is called.

This term is mentioned but not used in *Design Patterns*. However, the theme of the function object is repeated in a number of patterns in that book.

## Command: Choosing the Operation at Runtime

This is the function object in its purest sense: a method that's an object. By wrapping a method in an object, you can pass it to other methods or objects as a parameter, to tell them to perform this particular operation in the process of fulfilling your request:

```
# FunctionObjects/CommandPattern.py

class Command:
    def execute(self): pass

class Loony(Command):
    def execute(self):
        print("You're a loony.")

class NewBrain(Command):
    def execute(self):
        print("You might even need a new brain.")

class Afford(Command):
    def execute(self):
        print("I couldn't afford a whole new brain.")

# An object that holds commands:
class Macro:
```

```
    def __init__(self):
        self.commands = []
    def add(self, command):
        self.commands.append(command)
    def run(self):
        for c in self.commands:
            c.execute()

macro = Macro()
macro.add(Loony())
macro.add(NewBrain())
macro.add(Afford())
macro.run()
```

The primary point of *Command* is to allow you to hand a desired action to a method or object. In the above example, this provides a way to queue a set of actions to be performed collectively. In this case, it allows you to dynamically create new behavior, something you can normally only do by writing new code but in the above example could be done by interpreting a script (see the *Interpreter* pattern if what you need to do gets very complex).

*Design Patterns* says that "Commands are an object-oriented replacement for callbacks." However, I think that the word "back" is an essential part of the concept of callbacks. That is, I think a callback actually reaches back to the creator of the callback. On the other hand, with a *Command* object you typically just create it and hand it to some method or object, and are not otherwise connected over time to the *Command* object. That's my take on it, anyway. Later in this book, I combine a group of design patterns under the heading of "callbacks."

## Strategy: Choosing the Algorithm at Runtime

*Strategy* appears to be a family of *Command* classes, all inherited from the same base. But if you look at *Command*, you'll see that it has the same structure: a hierarchy of function objects. The difference is in the way this hierarchy is used. As seen in **patternRefactoring:DirList.py**, you use *Command* to solve a particular problem-in that case, selecting files from a list. The "thing that stays the same" is the body of the method that's being called, and the part that varies is isolated in the function object. I would hazard to say that *Command* provides flexibility while you're writing the program, whereas *Strategy*'s flexibility is at run time.

*Strategy* also adds a "Context" which can be a surrogate class that controls the selection and use of the particular strategy object-just like *State*! Here's what it looks like:

```
# FunctionObjects/StrategyPattern.py

# The strategy interface:
class FindMinima:
    # Line is a sequence of points:
    def algorithm(self, line) : pass

# The various strategies:
class LeastSquares(FindMinima):
    def algorithm(self, line):
        return [ 1.1, 2.2 ] # Dummy

class NewtonsMethod(FindMinima):
    def algorithm(self, line):
        return [ 3.3, 4.4 ]  # Dummy
```

```
class Bisection(FindMinima):
    def algorithm(self, line):
        return [ 5.5, 6.6 ] # Dummy

class ConjugateGradient(FindMinima):
    def algorithm(self, line):
        return [ 3.3, 4.4 ] # Dummy

# The "Context" controls the strategy:
class MinimaSolver:
    def __init__(self, strategy):
        self.strategy = strategy

    def minima(self, line):
        return self.strategy.algorithm(line)

    def changeAlgorithm(self, newAlgorithm):
        self.strategy = newAlgorithm

solver = MinimaSolver(LeastSquares())
line = [1.0, 2.0, 1.0, 2.0, -1.0, 3.0, 4.0, 5.0, 4.0]
print(solver.minima(line))
solver.changeAlgorithm(Bisection())
print(solver.minima(line))
```

Note similarity with template method - TM claims distinction that it has more than one method to call, does things piecewise. However, it's not unlikely that strategy object would have more than one method call; consider Shalloway's order fulfullment system with country information in each strategy.

Strategy example from standard Python: **sort( )** takes a second optional argument that acts as a comparator object; this is a strategy.

---

**Note:** A better, real world example is numerical integration, shown here: http://www.rosettacode.org/wiki/Numerical_Integration#Python

---

## Chain of Responsibility

*Chain of Responsibility* might be thought of as a dynamic generalization of recursion using *Strategy* objects. You make a call, and each *Strategy* in a linked sequence tries to satisfy the call. The process ends when one of the strategies is successful or the chain ends. In recursion, one method calls itself over and over until a termination condition is reached; with *Chain of Responsibility*, a method calls itself, which (by moving down the chain of *Strategies*) calls a different implementation of the method, etc., until a termination condition is reached. The termination condition is either the bottom of the chain is reached (in which case a default object is returned; you may or may not be able to provide a default result so you must be able to determine the success or failure of the chain) or one of the *Strategies* is successful.

Instead of calling a single method to satisfy a request, multiple methods in the chain have a chance to satisfy the request, so it has the flavor of an expert system. Since the chain is effectively a linked list, it can be dynamically created, so you could also think of it as a more general, dynamically-built **switch** statement.

In the GoF, there's a fair amount of Thidiscussion of how to create the chain of responsibility as a linked list. However, when you look at the pattern it really shouldn't matter how the chain is maintained; that's

an implementation detail. Since GoF was written before the Standard Template Library (STL) was incorporated into most C++ compilers, the reason for this is most likely (1) there was no list and thus they had to create one and (2) data structures are often taught as a fundamental skill in academia, and the idea that data structures should be standard tools available with the programming language may not have occurred to the GoF authors. I maintain that the implementation of *Chain of Responsibility* as a chain (specifically, a linked list) adds nothing to the solution and can just as easily be implemented using a standard Python list, as shown below. Furthermore, you'll see that I've gone to some effort to separate the chain-management parts of the implementation from the various *Strategies*, so that the code can be more easily reused.

In **StrategyPattern.py**, above, what you probably want is to automatically find a solution. *Chain of Responsibility* provides a way to do this by chaining the *Strategy* objects together and providing a mechanism for them to automatically recurse through each one in the chain:

```
# FunctionObjects/ChainOfResponsibility.py

# Carry the information into the strategy:
class Messenger: pass

# The Result object carries the result data and
# whether the strategy was successful:
class Result:
    def __init__(self):
        self.succeeded = 0
    def isSuccessful(self):
        return self.succeeded
    def setSuccessful(self, succeeded):
        self.succeeded = succeeded

class Strategy:
    def __call__(messenger): pass
    def __str__(self):
        return "Trying " + self.__class__.__name__ \
          + " algorithm"

# Manage the movement through the chain and
# find a successful result:
class ChainLink:
    def __init__(self, chain, strategy):
        self.strategy = strategy
        self.chain = chain
        self.chain.append(self)

    def next(self):
        # Where this link is in the chain:
        location = self.chain.index(self)
        if not self.end():
            return self.chain[location + 1]

    def end(self):
        return (self.chain.index(self) + 1 >=
                len(self.chain))

    def __call__(self, messenger):
        r = self.strategy(messenger)
        if r.isSuccessful() or self.end(): return r
        return self.next()(messenger)

# For this example, the Messenger
```

```
# and Result can be the same type:
class LineData(Result, Messenger):
    def __init__(self, data):
        self.data = data
    def __str__(self): return `self.data`

class LeastSquares(Strategy):
    def __call__(self, messenger):
        print(self)
        linedata = messenger
        # [ Actual test/calculation here ]
        result = LineData([1.1, 2.2]) # Dummy data
        result.setSuccessful(0)
        return result

class NewtonsMethod(Strategy):
    def __call__(self, messenger):
        print(self)
        linedata = messenger
        # [ Actual test/calculation here ]
        result = LineData([3.3, 4.4]) # Dummy data
        result.setSuccessful(0)
        return result

class Bisection(Strategy):
    def __call__(self, messenger):
        print(self)
        linedata = messenger
        # [ Actual test/calculation here ]
        result = LineData([5.5, 6.6]) # Dummy data
        result.setSuccessful(1)
        return result

class ConjugateGradient(Strategy):
    def __call__(self, messenger):
        print(self)
        linedata = messenger
        # [ Actual test/calculation here ]
        result = LineData([7.7, 8.8]) # Dummy data
        result.setSuccessful(1)
        return result

solutions = []
ChainLink(solutions, LeastSquares()),
ChainLink(solutions, NewtonsMethod()),
ChainLink(solutions, Bisection()),
ChainLink(solutions, ConjugateGradient())

line = LineData([
  1.0, 2.0, 1.0, 2.0, -1.0,
  3.0, 4.0, 5.0, 4.0
])

print(solutions[0](line))
```

## Exercises

1. Use *Command* in Chapter 3, Exercise 1.
2. Implement *Chain of Responsibility* to create an "expert system" that solves problems by successively trying one solution after another until one matches. You should be able to dynamically add solutions to the expert system. The test for solution should just be a string match, but when a solution fits, the expert system should return the appropriate type of **ProblemSolver** object. What other pattern/patterns show up here?

CHAPTER 34

# Changing the Interface

Sometimes the problem that you're solving is as simple as "I don't have the interface that I want." Two of the patterns in *Design Patterns* solve this problem: *Adapter* takes one type and produces an interface to some other type. *Façade* creates an interface to a set of classes, simply to provide a more comfortable way to deal with a library or bundle of resources.

## Adapter

When you've got *this*, and you need *that*, *Adapter* solves the problem. The only requirement is to produce a *that*, and there are a number of ways you can accomplish this adaptation:

```
# ChangeInterface/Adapter.py
# Variations on the Adapter pattern.

class WhatIHave:
    def g(self): pass
    def h(self): pass

class WhatIWant:
    def f(self): pass

class ProxyAdapter(WhatIWant):
    def __init__(self, whatIHave):
        self.whatIHave = whatIHave

    def f(self):
        # Implement behavior using
        # methods in WhatIHave:
        self.whatIHave.g()
        self.whatIHave.h()

class WhatIUse:
    def op(self, whatIWant):
```

```
        whatIWant.f()

# Approach 2: build adapter use into op():
class WhatIUse2(WhatIUse):
    def op(self, whatIHave):
        ProxyAdapter(whatIHave).f()

# Approach 3: build adapter into WhatIHave:
class WhatIHave2(WhatIHave, WhatIWant):
    def f(self):
        self.g()
        self.h()

# Approach 4: use an inner class:
class WhatIHave3(WhatIHave):
    class InnerAdapter(WhatIWant):
        def __init__(self, outer):
            self.outer = outer
        def f(self):
            self.outer.g()
            self.outer.h()

    def whatIWant(self):
        return WhatIHave3.InnerAdapter(self)

whatIUse = WhatIUse()
whatIHave = WhatIHave()
adapt= ProxyAdapter(whatIHave)
whatIUse2 = WhatIUse2()
whatIHave2 = WhatIHave2()
whatIHave3 = WhatIHave3()
whatIUse.op(adapt)
# Approach 2:
whatIUse2.op(whatIHave)
# Approach 3:
whatIUse.op(whatIHave2)
# Approach 4:
whatIUse.op(whatIHave3.whatIWant())
```

I'm taking liberties with the term "proxy" here, because in *Design Patterns* they assert that a proxy must have an identical interface with the object that it is a surrogate for. However, if you have the two words together: "proxy adapter," it is perhaps more reasonable.

## Façade

A general principle that I apply when I'm casting about trying to mold requirements into a first-cut object is "If something is ugly, hide it inside an object." This is basically what *Façade* accomplishes. If you have a rather confusing collection of classes and interactions that the client programmer doesn't really need to see, then you can create an interface that is useful for the client programmer and that only presents what's necessary.

Façade is often implemented as singleton abstract factory. Of course, you can easily get this effect by creating a class containing **static** factory methods:

```
# ChangeInterface/Facade.py
class A:
    def __init__(self, x): pass
class B:
    def __init__(self, x): pass
class C:
    def __init__(self, x): pass

# Other classes that aren't exposed by the
# facade go here ...

class Facade:
    def makeA(x): return A(x)
    makeA = staticmethod(makeA)
    def makeB(x): return B(x)
    makeB = staticmethod(makeB)
    def makeC(x): return C(x)
    makeC = staticmethod(makeC)

# The client programmer gets the objects
# by calling the static methods:
a = Facade.makeA(1);
b = Facade.makeB(1);
c = Facade.makeC(1.0);
```

[rewrite this section using research from Larman's book]

Example for Facade (?): my "nicer" version of the XML library.

## Exercises

1. Create an adapter class that automatically loads a two-dimensional array of objects into a dictionary as key-value pairs.

CHAPTER 35

# Table-Driven Code: Configuration Flexibility

## Table-Driven Code Using Anonymous Inner Classes

See **ListPerformance** example in TIJ from Chapter 9

Also **GreenHouse.py**

# CHAPTER 36

# Observer

Decoupling code behavior

*Observer*, and a category of callbacks called "multiple dispatching (not in *Design Patterns*)" including the *Visitor* from *Design Patterns*. Like the other forms of callback, this contains a hook point where you can change code. The difference is in the observer's completely dynamic nature. It is often used for the specific case of changes based on other object's change of state, but is also the basis of event management. Anytime you want to decouple the source of the call from the called code in a completely dynamic way.

The observer pattern solves a fairly common problem: What if a group of objects needs to update themselves when some object changes state? This can be seen in the "model-view" aspect of Smalltalk's MVC (model-view-controller), or the almost-equivalent "Document-View Architecture." Suppose that you have some data (the "document") and more than one view, say a plot and a textual view. When you change the data, the two views must know to update themselves, and that's what the observer facilitates. It's a common enough problem that its solution has been made a part of the standard **java.util** library.

There are two types of objects used to implement the observer pattern in Python. The **Observable** class keeps track of everybody who wants to be informed when a change happens, whether the "state" has changed or not. When someone says "OK, everybody should check and potentially update themselves," the **Observable** class performs this task by calling the **notifyObservers( )** method for each one on the list. The **notifyObservers( )** method is part of the base class **Observable**.

There are actually two "things that change" in the observer pattern: the quantity of observing objects and the way an update occurs. That is, the observer pattern allows you to modify both of these without affecting the surrounding code.

**Observer** is an "interface" class that only has one member function, **update( )**. This function is called by the object that's being observed, when that object decides its time to update all its observers. The arguments are optional; you could have an **update( )** with no arguments and that would still fit the observer pattern; however this is more general-it allows the observed object to pass the object that caused the update (since an **Observer** may be registered with more than one observed object) and any extra information if that's helpful, rather than forcing the **Observer** object to hunt around to see who is updating and to fetch any other information it needs.

The "observed object" that decides when and how to do the updating will be called the **Observable**.

**Observable** has a flag to indicate whether it's been changed. In a simpler design, there would be no flag; if something happened, everyone would be notified. The flag allows you to wait, and only notify the **Observer**s when you decide the time is right. Notice, however, that the control of the flag's state is **protected**, so that only an inheritor can decide what constitutes a change, and not the end user of the resulting derived **Observer** class.

Most of the work is done in **notifyObservers( )**. If the **changed** flag has not been set, this does nothing. Otherwise, it first clears the **changed** flag so repeated calls to **notifyObservers( )** won't waste time. This is done before notifying the observers in case the calls to **update( )** do anything that causes a change back to this **Observable** object. Then it moves through the **set** and calls back to the **update( )** member function of each **Observer**.

At first it may appear that you can use an ordinary **Observable** object to manage the updates. But this doesn't work; to get an effect, you *must* inherit from **Observable** and somewhere in your derived-class code call **setChanged( )**. This is the member function that sets the "changed" flag, which means that when you call **notifyObservers( )** all of the observers will, in fact, get notified. *Where* you call **setChanged( )** depends on the logic of your program.

## Observing Flowers

Since Python doesn't have standard library components to support the observer pattern (like Java does), we must first create one. The simplest thing to do is translate the Java standard library **Observer** and **Observable** classes. This also provides easier translation from Java code that uses these libraries.

In trying to do this, we encounter a minor snag, which is the fact that Java has a **synchronized** keyword that provides built-in support for thread synchronization. We could certainly accomplish the same thing by hand, using code like this:

```
# Util/ToSynch.py

import threading
class ToSynch:
    def __init__(self):
        self.mutex = threading.RLock()
        self.val = 1
    def aSynchronizedMethod(self):
        self.mutex.acquire()
        try:
            self.val += 1
            return self.val
        finally:
            self.mutex.release()
```

But this rapidly becomes tedious to write and to read. Peter Norvig provided me with a much nicer solution:

```
# Util/Synchronization.py
'''Simple emulation of Java's 'synchronized'
keyword, from Peter Norvig.'''
import threading

def synchronized(method):
    def f(*args):
        self = args[0]
        self.mutex.acquire();
        # print(method.__name__, 'acquired')
```

```
        try:
            return apply(method, args)
        finally:
            self.mutex.release();
            # print(method.__name__, 'released')
    return f

def synchronize(klass, names=None):
    """Synchronize methods in the given class.
    Only synchronize the methods whose names are
    given, or all methods if names=None."""
    if type(names)==type(''): names = names.split()
    for (name, val) in klass.__dict__.items():
        if callable(val) and name != '__init__' and \
          (names == None or name in names):
            # print("synchronizing", name)
            klass.__dict__[name] = synchronized(val)

# You can create your own self.mutex, or inherit
# from this class:
class Synchronization:
    def __init__(self):
        self.mutex = threading.RLock()
```

The **synchronized( )** function takes a method and wraps it in a function that adds the mutex functionality. The method is called inside this function:

```
return apply(method, args)
```

and as the **return** statement passes through the **finally** clause, the mutex is released.

This is in some ways the *Decorator* design pattern, but much simpler to create and use. All you have to say is:

```
myMethod = synchronized(myMethod)
```

To surround your method with a mutex.

**synchronize( )** is a convenience function that applies **synchronized( )** to an entire class, either all the methods in the class (the default) or selected methods which are named in a string as the second argument.

Finally, for **synchronized( )** to work there must be a **self.mutex** created in every class that uses **synchronized( )**. This can be created by hand by the class author, but it's more consistent to use inheritance, so the base class **Synchronization** is provided.

Here's a simple test of the **Synchronization** module:

```
# Util/TestSynchronization.py
from Synchronization import *

# To use for a method:
class C(Synchronization):
    def __init__(self):
        Synchronization.__init__(self)
        self.data = 1
    def m(self):
        self.data += 1
        return self.data
    m = synchronized(m)
```

```
    def f(self): return 47
    def g(self): return 'spam'

# So m is synchronized, f and g are not.
c = C()

# On the class level:
class D(C):
    def __init__(self):
        C.__init__(self)
    # You must override an un-synchronized method
    # in order to synchronize it (just like Java):
    def f(self): C.f(self)

# Synchronize every (defined) method in the class:
synchronize(D)
d = D()
d.f() # Synchronized
d.g() # Not synchronized
d.m() # Synchronized (in the base class)

class E(C):
    def __init__(self):
        C.__init__(self)
    def m(self): C.m(self)
    def g(self): C.g(self)
    def f(self): C.f(self)
# Only synchronizes m and g. Note that m ends up
# being doubly-wrapped in synchronization, which
# doesn't hurt anything but is inefficient:
synchronize(E, 'm g')
e = E()
e.f()
e.g()
e.m()
```

You must call the base class constructor for **Synchronization**, but that's all. In class **C** you can see the use of **synchronized( )** for **m**, leaving **f** and **g** alone. Class **D** has all its methods synchronized en masse, and class **E** uses the convenience function to synchronize **m** and **g**. Note that since **m** ends up being synchronized twice, it will be entered and left twice for every call, which isn't very desirable [there may be a fix for this]:

```
# Util/Observer.py
# Class support for "observer" pattern.
from Synchronization import *

class Observer:
    def update(observable, arg):
        '''Called when the observed object is
        modified. You call an Observable object's
        notifyObservers method to notify all the
        object's observers of the change.'''
        pass

class Observable(Synchronization):
    def __init__(self):
        self.obs = []
        self.changed = 0
        Synchronization.__init__(self)
```

```
    def addObserver(self, observer):
        if observer not in self.obs:
            self.obs.append(observer)

    def deleteObserver(self, observer):
        self.obs.remove(observer)

    def notifyObservers(self, arg = None):
        '''If 'changed' indicates that this object
        has changed, notify all its observers, then
        call clearChanged(). Each observer has its
        update() called with two arguments: this
        observable object and the generic 'arg'.'''

        self.mutex.acquire()
        try:
            if not self.changed: return
            # Make a local copy in case of synchronous
            # additions of observers:
            localArray = self.obs[:]
            self.clearChanged()
        finally:
            self.mutex.release()
        # Updating is not required to be synchronized:
        for observer in localArray:
            observer.update(self, arg)

    def deleteObservers(self): self.obs = []
    def setChanged(self): self.changed = 1
    def clearChanged(self): self.changed = 0
    def hasChanged(self): return self.changed
    def countObservers(self): return len(self.obs)

synchronize(Observable,
  "addObserver deleteObserver deleteObservers " +
  "setChanged clearChanged hasChanged " +
  "countObservers")
```

Using this library, here is an example of the observer pattern:

```
# Observer/ObservedFlower.py
# Demonstration of "observer" pattern.
import sys
sys.path += ['../util']
from Observer import Observer, Observable

class Flower:
    def __init__(self):
        self.isOpen = 0
        self.openNotifier = Flower.OpenNotifier(self)
        self.closeNotifier= Flower.CloseNotifier(self)
    def open(self): # Opens its petals
        self.isOpen = 1
        self.openNotifier.notifyObservers()
        self.closeNotifier.open()
    def close(self): # Closes its petals
        self.isOpen = 0
```

```
        self.closeNotifier.notifyObservers()
        self.openNotifier.close()
    def closing(self): return self.closeNotifier

    class OpenNotifier(Observable):
        def __init__(self, outer):
            Observable.__init__(self)
            self.outer = outer
            self.alreadyOpen = 0
        def notifyObservers(self):
            if self.outer.isOpen and \
            not self.alreadyOpen:
                self.setChanged()
                Observable.notifyObservers(self)
                self.alreadyOpen = 1
        def close(self):
            self.alreadyOpen = 0

    class CloseNotifier(Observable):
        def __init__(self, outer):
            Observable.__init__(self)
            self.outer = outer
            self.alreadyClosed = 0
        def notifyObservers(self):
            if not self.outer.isOpen and \
            not self.alreadyClosed:
                self.setChanged()
                Observable.notifyObservers(self)
                self.alreadyClosed = 1
        def open(self):
            alreadyClosed = 0

class Bee:
    def __init__(self, name):
        self.name = name
        self.openObserver = Bee.OpenObserver(self)
        self.closeObserver = Bee.CloseObserver(self)
    # An inner class for observing openings:
    class OpenObserver(Observer):
        def __init__(self, outer):
            self.outer = outer
        def update(self, observable, arg):
            print("Bee " + self.outer.name + \)
              "'s breakfast time!"
    # Another inner class for closings:
    class CloseObserver(Observer):
        def __init__(self, outer):
            self.outer = outer
        def update(self, observable, arg):
            print("Bee " + self.outer.name + \)
              "'s bed time!"

class Hummingbird:
    def __init__(self, name):
        self.name = name
        self.openObserver = \
          Hummingbird.OpenObserver(self)
        self.closeObserver = \
```

```
            Hummingbird.CloseObserver(self)
    class OpenObserver(Observer):
        def __init__(self, outer):
            self.outer = outer
        def update(self, observable, arg):
            print("Hummingbird " + self.outer.name + \
              "'s breakfast time!")
    class CloseObserver(Observer):
        def __init__(self, outer):
            self.outer = outer
        def update(self, observable, arg):
            print("Hummingbird " + self.outer.name + \
              "'s bed time!")

f = Flower()
ba = Bee("Eric")
bb = Bee("Eric 0.5")
ha = Hummingbird("A")
hb = Hummingbird("B")
f.openNotifier.addObserver(ha.openObserver)
f.openNotifier.addObserver(hb.openObserver)
f.openNotifier.addObserver(ba.openObserver)
f.openNotifier.addObserver(bb.openObserver)
f.closeNotifier.addObserver(ha.closeObserver)
f.closeNotifier.addObserver(hb.closeObserver)
f.closeNotifier.addObserver(ba.closeObserver)
f.closeNotifier.addObserver(bb.closeObserver)
# Hummingbird 2 decides to sleep in:
f.openNotifier.deleteObserver(hb.openObserver)
# A change that interests observers:
f.open()
f.open() # It's already open, no change.
# Bee 1 doesn't want to go to bed:
f.closeNotifier.deleteObserver(ba.closeObserver)
f.close()
f.close() # It's already closed; no change
f.openNotifier.deleteObservers()
f.open()
f.close()
```

The events of interest are that a **Flower** can open or close. Because of the use of the inner class idiom, both these events can be separately observable phenomena. **OpenNotifier** and **CloseNotifier** both inherit **Observable**, so they have access to **setChanged( )** and can be handed to anything that needs an **Observable**.

The inner class idiom also comes in handy to define more than one kind of **Observer**, in **Bee** and **Hummingbird**, since both those classes may want to independently observe **Flower** openings and closings. Notice how the inner class idiom provides something that has most of the benefits of inheritance (the ability to access the **private** data in the outer class, for example) without the same restrictions.

In **main( )**, you can see one of the prime benefits of the observer pattern: the ability to change behavior at run time by dynamically registering and un- registering **Observer**s with **Observable**s.

If you study the code above you'll see that **OpenNotifier** and **CloseNotifier** use the basic **Observable** interface. This means that you could inherit other completely different **Observer** classes; the only connection the **Observer**s have with **Flower**s is the **Observer** interface.

## A Visual Example of Observers

The following example is similar to the **ColorBoxes** example from *Thinking in Java*. Boxes are placed in a grid on the screen and each one is initialized to a random color. In addition, each box **implements** the **Observer** interface and is registered with an **Observable** object. When you click on a box, all of the other boxes are notified that a change has been made because the **Observable** object automatically calls each **Observer** object's **update( )** method. Inside this method, the box checks to see if it's adjacent to the one that was clicked, and if so it changes its color to match the clicked box. (NOTE: this example has not been converted. See further down for a version that has the GUI but not the Observers, in PythonCard.):

```
# Observer/BoxObserver.py
# Demonstration of Observer pattern using
# Java's built-in observer classes.

# You must inherit a type of Observable:
class BoxObservable(Observable):
    def notifyObservers(self, Object b):
        # Otherwise it won't propagate changes:
        setChanged()
        super.notifyObservers(b)

class BoxObserver(JFrame):
    Observable notifier = BoxObservable()
    def __init__(self, grid):
        setTitle("Demonstrates Observer pattern")
        Container cp = getContentPane()
        cp.setLayout(GridLayout(grid, grid))
        for(int x = 0 x < grid x++)
            for(int y = 0 y < grid y++)
                cp.add(OCBox(x, y, notifier))

    def main(self, String[] args):
        grid = 8
            if(args.length > 0)
                grid = Integer.parseInt(args[0])
            JFrame f = BoxObserver(grid)
            f.setSize(500, 400)
            f.setVisible(1)
            # JDK 1.3:
            f.setDefaultCloseOperation(EXIT_ON_CLOSE)
            # Add a WindowAdapter if you have JDK 1.2

class OCBox(JPanel) implements Observer:
    Color cColor = newColor()
    colors = [
      Color.black, Color.blue, Color.cyan,
      Color.darkGray, Color.gray, Color.green,
      Color.lightGray, Color.magenta,
      Color.orange, Color.pink, Color.red,
      Color.white, Color.yellow
    ]
    def newColor():
        return colors[
          (int)(Math.random() * colors.length)
        ]

    def __init__(self, x, y, Observable notifier):
        self.x = x
```

```
        self.y = y
        notifier.addObserver(self)
        self.notifier = notifier
        addMouseListener(ML())

    def paintComponent(self, Graphics g):
        super.paintComponent(g)
        g.setColor(cColor)
        Dimension s = getSize()
        g.fillRect(0, 0, s.width, s.height)

    class ML(MouseAdapter):
        def mousePressed(self, MouseEvent e):
            notifier.notifyObservers(OCBox.self)

    def update(self, Observable o, Object arg):
        OCBox clicked = (OCBox)arg
        if(nextTo(clicked)):
            cColor = clicked.cColor
            repaint()

    def nextTo(OCBox b):
        return Math.abs(x - b.x) <= 1 &&
            Math.abs(y - b.y) <= 1
```

When you first look at the online documentation for **Observable**, it's a bit confusing because it appears that you can use an ordinary **Observable** object to manage the updates. But this doesn't work; try it-inside **BoxObserver**, create an **Observable** object instead of a **BoxObservable** object and see what happens: nothing. To get an effect, you *must* inherit from **Observable** and somewhere in your derived-class code call **setChanged( )**. This is the method that sets the "changed" flag, which means that when you call **notifyObservers( )** all of the observers will, in fact, get notified. In the example above **setChanged( )** is simply called within **notifyObservers( )**, but you could use any criterion you want to decide when to call **setChanged( )**.

**BoxObserver** contains a single **Observable** object called **notifier**, and every time an **OCBox** object is created, it is tied to **notifier**. In **OCBox**, whenever you click the mouse the **notifyObservers( )** method is called, passing the clicked object in as an argument so that all the boxes receiving the message (in their **update( )** method) know who was clicked and can decide whether to change themselves or not. Using a combination of code in **notifyObservers( )** and **update( )** you can work out some fairly complex schemes.

It might appear that the way the observers are notified must be frozen at compile time in the **notifyObservers( )** method. However, if you look more closely at the code above you'll see that the only place in **BoxObserver** or **OCBox** where you're aware that you're working with a **BoxObservable** is at the point of creation of the **Observable** object-from then on everything uses the basic **Observable** interface. This means that you could inherit other **Observable** classes and swap them at run time if you want to change notification behavior then.

Here is a version of the above that doesn't use the Observer pattern, written by Kevin Altis using PythonCard, and placed here as a starting point for a translation that does include Observer:

```
# Observer/BoxObserverPythonCard.py
""" Written by Kevin Altis as a first-cut for
converting BoxObserver to Python. The Observer
hasn't been integrated yet.
To run this program, you must:
Install WxPython from
http://www.wxpython.org/download.php
Install PythonCard. See:
http://pythoncard.sourceforge.net
```

```
"""
from PythonCardPrototype import log, model
import random

GRID = 8

class ColorBoxesTest(model.Background):
    def on_openBackground(self, event):
        self.document = []
        for row in range(GRID):
            line = []
            for column in range(GRID):
                line.append(self.createBox(row, column))
            self.document.append(line[:])
    def createBox(self, row, column):
        colors = ['black', 'blue', 'cyan',
        'darkGray', 'gray', 'green',
        'lightGray', 'magenta',
        'orange', 'pink', 'red',
        'white', 'yellow']
        width, height = self.panel.GetSizeTuple()
        boxWidth = width / GRID
        boxHeight = height / GRID
        log.info("width:" + str(width) +
          " height:" + str(height))
        log.info("boxWidth:" + str(boxWidth) +
          " boxHeight:" + str(boxHeight))
        # use an empty image, though some other
        # widgets would work just as well
        boxDesc = {'type':'Image',
          'size':(boxWidth, boxHeight), 'file':''}
        name = 'box-%d-%d' % (row, column)
        # There is probably a 1 off error in the
        # calculation below since the boxes should
        # probably have a slightly different offset
        # to prevent overlaps
        boxDesc['position'] = \
          (column * boxWidth, row * boxHeight)
        boxDesc['name'] = name
        boxDesc['backgroundColor'] = \
          random.choice(colors)
        self.components[name] =  boxDesc
        return self.components[name]

    def changeNeighbors(self, row, column, color):

        # This algorithm will result in changing the
        # color of some boxes more than once, so an
        # OOP solution where only neighbors are asked
        # to change or boxes check to see if they are
        # neighbors before changing would be better
        # per the original example does the whole grid
        # need to change its state at once like in a
        # Life program? should the color change
        # in the propogation of another notification
        # event?

        for r in range(max(0, row - 1),
```

```
                        min(GRID, row + 2)):
                for c in range(max(0, column - 1),
                               min(GRID, column + 2)):
                    self.document[r][c].backgroundColor=color

    # this is a background handler, so it isn't
    # specific to a single widget. Image widgets
    # don't have a mouseClick event (wxCommandEvent
    # in wxPython)
    def on_mouseUp(self, event):
        target = event.target
        prefix, row, column = target.name.split('-')
        self.changeNeighbors(int(row), int(column),
                             target.backgroundColor)

if __name__ == '__main__':
    app = model.PythonCardApp(ColorBoxesTest)
    app.MainLoop()
```

This is the resource file for running the program (see PythonCard for details):

```
# Observer/BoxObserver.rsrc.py
{'stack':{'type':'Stack',
          'name':'BoxObserver',
    'backgrounds': [
      { 'type':'Background',
        'name':'bgBoxObserver',
        'title':'Demonstrates Observer pattern',
        'position':(5, 5),
        'size':(500, 400),
        'components': [

] # end components
} # end background
] # end backgrounds
} }
```

## Exercises

1. Using the approach in **Synchronization.py**, create a tool that will automatically wrap all the methods in a class to provide an execution trace, so that you can see the name of the method and when it is entered and exited.
2. Create a minimal Observer-Observable design in two classes. Just create the bare minimum in the two classes, then demonstrate your design by creating one **Observable** and many **Observer**s, and cause the **Observable** to update the **Observer**s.
3. Modify **BoxObserver.py** to turn it into a simple game. If any of the squares surrounding the one you clicked is part of a contiguous patch of the same color, then all the squares in that patch are changed to the color you clicked on. You can configure the game for competition between players or to keep track of the number of clicks that a single player uses to turn the field into a single color. You may also want to restrict a player's color to the first one that was chosen.

CHAPTER 37

# Multiple Dispatching

When dealing with multiple types which are interacting, a program can get particularly messy. For example, consider a system that parses and executes mathematical expressions. You want to be able to say **Number + Number**, **Number * Number**, etc., where **Number** is the base class for a family of numerical objects. But when you say **a + b**, and you don't know the exact type of either **a** or **b**, so how can you get them to interact properly?

The answer starts with something you probably don't think about: Python performs only single dispatching. That is, if you are performing an operation on more than one object whose type is unknown, Python can invoke the dynamic binding mechanism on only one of those types. This doesn't solve the problem, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

The solution is called *multiple dispatching*. Remember that polymorphism can occur only via member function calls, so if you want double dispatching to occur, there must be two member function calls: the first to determine the first unknown type, and the second to determine the second unknown type. With multiple dispatching, you must have a polymorphic method call to determine each of the types. Generally, you'll set up a configuration such that a single member function call produces more than one dynamic member function call and thus determines more than one type in the process. To get this effect, you need to work with more than one polymorphic method call: you'll need one call for each dispatch. The methods in the following example are called **compete( )** and **eval( )**, and are both members of the same type. (In this case there will be only two dispatches, which is referred to as *double dispatching*). If you are working with two different type hierarchies that are interacting, then you'll have to have a polymorphic method call in each hierarchy.

Here's an example of multiple dispatching:

```
# MultipleDispatching/PaperScissorsRock.py
# Demonstration of multiple dispatching.
from __future__ import generators
import random

# An enumeration type:
class Outcome:
    def __init__(self, value, name):
        self.value = value
```

```
        self.name = name
    def __str__(self): return self.name
    def __eq__(self, other):
        return self.value == other.value

Outcome.WIN = Outcome(0, "win")
Outcome.LOSE = Outcome(1, "lose")
Outcome.DRAW = Outcome(2, "draw")

class Item(object):
    def __str__(self):
        return self.__class__.__name__

class Paper(Item):
    def compete(self, item):
        # First dispatch: self was Paper
        return item.evalPaper(self)
    def evalPaper(self, item):
        # Item was Paper, we're in Paper
        return Outcome.DRAW
    def evalScissors(self, item):
        # Item was Scissors, we're in Paper
        return Outcome.WIN
    def evalRock(self, item):
        # Item was Rock, we're in Paper
        return Outcome.LOSE

class Scissors(Item):
    def compete(self, item):
        # First dispatch: self was Scissors
        return item.evalScissors(self)
    def evalPaper(self, item):
        # Item was Paper, we're in Scissors
        return Outcome.LOSE
    def evalScissors(self, item):
        # Item was Scissors, we're in Scissors
        return Outcome.DRAW
    def evalRock(self, item):
        # Item was Rock, we're in Scissors
        return Outcome.WIN

class Rock(Item):
    def compete(self, item):
        # First dispatch: self was Rock
        return item.evalRock(self)
    def evalPaper(self, item):
        # Item was Paper, we're in Rock
        return Outcome.WIN
    def evalScissors(self, item):
        # Item was Scissors, we're in Rock
        return Outcome.LOSE
    def evalRock(self, item):
        # Item was Rock, we're in Rock
        return Outcome.DRAW

def match(item1, item2):
    print("%s <--> %s : %s" % (
      item1, item2, item1.compete(item2)))
```

```
# Generate the items:
def itemPairGen(n):
    # Create a list of instances of all Items:
    Items = Item.__subclasses__()
    for i in range(n):
        yield (random.choice(Items)(),
               random.choice(Items)())

for item1, item2 in itemPairGen(20):
    match(item1, item2)
```

This was a fairly literal translation from the Java version, and one of the things you might notice is that the information about the various combinations is encoded into each type of **Item**. It actually ends up being a kind of table, except that it is spread out through all the classes. This is not very easy to maintain if you ever expect to modify the behavior or to add a new **Item** class. Instead, it can be more sensible to make the table explicit, like this:

```
# MultipleDispatching/PaperScissorsRock2.py
# Multiple dispatching using a table
from __future__ import generators
import random

class Outcome:
    def __init__(self, value, name):
        self.value = value
        self.name = name
    def __str__(self): return self.name
    def __eq__(self, other):
        return self.value == other.value

Outcome.WIN = Outcome(0, "win")
Outcome.LOSE = Outcome(1, "lose")
Outcome.DRAW = Outcome(2, "draw")

class Item(object):
    def compete(self, item):
        # Use a tuple for table lookup:
        return outcome[self.__class__, item.__class__]
    def __str__(self):
        return self.__class__.__name__

class Paper(Item): pass
class Scissors(Item): pass
class Rock(Item): pass

outcome = {
  (Paper, Rock): Outcome.WIN,
  (Paper, Scissors): Outcome.LOSE,
  (Paper, Paper): Outcome.DRAW,
  (Scissors, Paper): Outcome.WIN,
  (Scissors, Rock): Outcome.LOSE,
  (Scissors, Scissors): Outcome.DRAW,
  (Rock, Scissors): Outcome.WIN,
  (Rock, Paper): Outcome.LOSE,
  (Rock, Rock): Outcome.DRAW,
}
```

```
def match(item1, item2):
    print("%s <--> %s : %s" % (
      item1, item2, item1.compete(item2)))

# Generate the items:
def itemPairGen(n):
    # Create a list of instances of all Items:
    Items = Item.__subclasses__()
    for i in range(n):
        yield (random.choice(Items)(),
               random.choice(Items)())

for item1, item2 in itemPairGen(20):
    match(item1, item2)
```

It's a tribute to the flexibility of dictionaries that a tuple can be used as a key just as easily as a single object.

CHAPTER 38

# Visitor

The visitor pattern is implemented using multiple dispatching, but people often confuse the two, because they look at the implementation rather than the intent.

The assumption is that you have a primary class hierarchy that is fixed; perhaps it's from another vendor and you can't make changes to that hierarchy. However, your intent is that you'd like to add new polymorphic methods to that hierarchy, which means that normally you'd have to add something to the base class interface. So the dilemma is that you need to add methods to the base class, but you can't touch the base class. How do you get around this?

The design pattern that solves this kind of problem is called a "visitor" (the final one in the *Design Patterns* book), and it builds on the double dispatching scheme shown in the last section.

The visitor pattern allows you to extend the interface of the primary type by creating a separate class hierarchy of type **Visitor** to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound member function:

```
# Visitor/FlowerVisitors.py
# Demonstration of "visitor" pattern.
from __future__ import generators
import random

# The Flower hierarchy cannot be changed:
class Flower(object):
    def accept(self, visitor):
        visitor.visit(self)
    def pollinate(self, pollinator):
        print(self, "pollinated by", pollinator)
    def eat(self, eater):
        print(self, "eaten by", eater)
    def __str__(self):
        return self.__class__.__name__

class Gladiolus(Flower): pass
class Runuculus(Flower): pass
class Chrysanthemum(Flower): pass
```

```
class Visitor:
    def __str__(self):
        return self.__class__.__name__

class Bug(Visitor): pass
class Pollinator(Bug): pass
class Predator(Bug): pass

# Add the ability to do "Bee" activities:
class Bee(Pollinator):
    def visit(self, flower):
        flower.pollinate(self)

# Add the ability to do "Fly" activities:
class Fly(Pollinator):
    def visit(self, flower):
        flower.pollinate(self)

# Add the ability to do "Worm" activities:
class Worm(Predator):
    def visit(self, flower):
        flower.eat(self)

def flowerGen(n):
    flwrs = Flower.__subclasses__()
    for i in range(n):
        yield random.choice(flwrs)()

# It's almost as if I had a method to Perform
# various "Bug" operations on all Flowers:
bee = Bee()
fly = Fly()
worm = Worm()
for flower in flowerGen(10):
    flower.accept(bee)
    flower.accept(fly)
    flower.accept(worm)
```

## Exercises

1. Create a business-modeling environment with three types of **Inhabitant**: **Dwarf** (for engineers), **Elf** (for marketers) and **Troll** (for managers). Now create a class called **Project** that creates the different inhabitants and causes them to **interact( )** with each other using multiple dispatching.
2. Modify the above example to make the interactions more detailed. Each **Inhabitant** can randomly produce a **Weapon** using **getWeapon( )**: a **Dwarf** uses **Jargon** or **Play**, an **Elf** uses **InventFeature** or **SellImaginaryProduct**, and a **Troll** uses **Edict** and **Schedule**. You must decide which weapons "win" and "lose" in each interaction (as in **PaperScissorsRock.py**). Add a **battle( )** member function to **Project** that takes two **Inhabitant**s and matches them against each other. Now create a **meeting( )** member function for **Project** that creates groups of **Dwarf**, **Elf** and **Manager** and battles the groups against each other until only members of one group are left standing. These are the "winners."
3. Modify **PaperScissorsRock.py** to replace the double dispatching with a table lookup. The easiest way to do this is to create a **Map** of **Map**s, with the key of each **Map** the class of each object. Then you

can do the lookup by saying: *((Map)map.get(o1.getClass())).get(o2.getClass())* Notice how much easier it is to reconfigure the system. When is it more appropriate to use this approach vs. hard-coding the dynamic dispatches? Can you create a system that has the syntactic simplicity of use of the dynamic dispatch but uses a table lookup?

4. Modify Exercise 2 to use the table lookup technique described in Exercise 3.

CHAPTER 39

# Pattern Refactoring

**Note:** This chapter has not had any significant translation yet.

This chapter will look at the process of solving a problem by applying design patterns in an evolutionary fashion. That is, a first cut design will be used for the initial solution, and then this solution will be examined and various design patterns will be applied to the problem (some of which will work, and some of which won't). The key question that will always be asked in seeking improved solutions is "what will change?"

This process is similar to what Martin Fowler talks about in his book *Refactoring: Improving the Design of Existing Code [#]_* (although he tends to talk about pieces of code more than pattern-level designs). You start with a solution, and then when you discover that it doesn't continue to meet your needs, you fix it. Of course, this is a natural tendency but in computer programming it's been extremely difficult to accomplish with procedural programs, and the acceptance of the idea that we *can* refactor code and designs adds to the body of proof that object-oriented programming is "a good thing."

## Simulating the Trash Recycler

The nature of this problem is that the trash is thrown unclassified into a single bin, so the specific type information is lost. But later, the specific type information must be recovered to properly sort the trash. In the initial solution, RTTI (described in *Thinking in Java*) is used.

This is not a trivial design because it has an added constraint. That's what makes it interesting-it's more like the messy problems you're likely to encounter in your work. The extra constraint is that the trash arrives at the trash recycling plant all mixed together. The program must model the sorting of that trash. This is where RTTI comes in: you have a bunch of anonymous pieces of trash, and the program figures out exactly what type they are:

```
# PatternRefactoring/recyclea/RecycleA.py
# Recycling with RTTI.

class Trash:
```

```
    def __init__(self, wt):
        self.weight = wt
    abstract def getValue()
    def getWeight(): return weight
    # Sums the value of Trash in a bin:
    def sumValue(Iterator it):
        val = 0.0f
        while(it.hasNext()):
            # One kind of RTTI:
            # A dynamically-checked cast
            Trash t = (Trash)it.next()
            # Polymorphism in action:
            val += t.getWeight() * t.getValue()
            print (
              "weight of " +
              # Using RTTI to get type
              # information about the class:
              t.getClass().getName() +
              " = " + t.getWeight())

        print("Total value = " + val)

class Aluminum(Trash):
    val  = 1.67f
    def __init__(self, wt):
        Trash.__init__(wt)
    def getValue(): return self.val
    setValue(newval):
        val = newval

class Paper(Trash):
    val = 0.10f
    def __init__(self, wt):
        Trash.__init__(wt)
    def getValue(): return self.val
    def setValue(self, newval):
        val = newval

class Glass(Trash):
    val = 0.23f
    def __init__(self, wt):
        Trash.__init__(wt)
    def getValue(self):
        return self.val
    def setValue(self, newval):
        val = newval

class RecycleA(UnitTest):
    bin = ArrayList()
    glassBin = ArrayList()
    paperBin = ArrayList()
    alBin = ArrayList()
    def __init__(self):
        # Fill up the Trash bin:
        for(int i = 0 i < 30 i++)
            switch((int)(Math.random() * 3)):
                case 0:
                    bin.add(new
```

```
                    Aluminum(Math.random() * 100))
                break
            case 1:
                bin.add(new
                  Paper(Math.random() * 100))
                break
            case 2:
                bin.add(new
                  Glass(Math.random() * 100))

    def test(self):
        Iterator sorter = bin.iterator()
        # Sort the Trash:
        while(sorter.hasNext()):
            Object t = sorter.next()
            # RTTI to show class membership:
            if(t instanceof Aluminum)
                alBin.add(t)
            if(t instanceof Paper)
                paperBin.add(t)
            if(t instanceof Glass)
                glassBin.add(t)

        Trash.sumValue(alBin.iterator())
        Trash.sumValue(paperBin.iterator())
        Trash.sumValue(glassBin.iterator())
        Trash.sumValue(bin.iterator())

    def main(self, String args[]):
        RecycleA().test()
```

In the source code listings available for this book, this file will be placed in the subdirectory **recyclea** that branches off from the subdirectory **patternRefactoring**. The unpacking tool takes care of placing it into the correct subdirectory. The reason for doing this is that this chapter rewrites this particular example a number of times and by putting each version in its own directory (using the default package in each directory so that invoking the program is easy), the class names will not clash.

Several **ArrayList** objects are created to hold **Trash** references. Of course, **ArrayList**s actually hold **Object**s so they'll hold anything at all. The reason they hold **Trash** (or something derived from **Trash**) is only because you've been careful to not put in anything except **Trash**. If you do put something "wrong" into the **ArrayList**, you won't get any compile-time warnings or errors-you'll find out only via an exception at run time.

When the **Trash** references are added, they lose their specific identities and become simply **Object reference**s (they are *upcast*). However, because of polymorphism the proper behavior still occurs when the dynamically-bound methods are called through the **Iterator sorter**, once the resulting **Object** has been cast back to **Trash**. **sumValue( )** also takes an **Iterator** to perform operations on every object in the **ArrayList**.

It looks silly to upcast the types of **Trash** into a container holding base type references, and then turn around and downcast. Why not just put the trash into the appropriate receptacle in the first place? (Indeed, this is the whole enigma of recycling). In this program it would be easy to repair, but sometimes a system's structure and flexibility can benefit greatly from downcasting.

The program satisfies the design requirements: it works. This might be fine as long as it's a one-shot solution. However, a useful program tends to evolve over time, so you must ask, "What if the situation changes?" For example, cardboard is now a valuable recyclable commodity, so how will that be integrated into the system (especially if the program is large and complicated). Since the above type-check coding in the **switch** statement could be scattered throughout the program, you must go find all that code every time

a new type is added, and if you miss one the compiler won't give you any help by pointing out an error.

The key to the misuse of RTTI here is that *every type is tested*. If you're looking for only a subset of types because that subset needs special treatment, that's probably fine. But if you're hunting for every type inside a switch statement, then you're probably missing an important point, and definitely making your code less maintainable. In the next section we'll look at how this program evolved over several stages to become much more flexible. This should prove a valuable example in program design.

## Improving the Design

The solutions in *Design Patterns* are organized around the question "What will change as this program evolves?" This is usually the most important question that you can ask about any design. If you can build your system around the answer, the results will be two-pronged: not only will your system allow easy (and inexpensive) maintenance, but you might also produce components that are reusable, so that other systems can be built more cheaply. This is the promise of object-oriented programming, but it doesn't happen automatically; it requires thought and insight on your part. In this section we'll see how this process can happen during the refinement of a system.

The answer to the question "What will change?" for the recycling system is a common one: more types will be added to the system. The goal of the design, then, is to make this addition of types as painless as possible. In the recycling program, we'd like to encapsulate all places where specific type information is mentioned, so (if for no other reason) any changes can be localized to those encapsulations. It turns out that this process also cleans up the rest of the code considerably.

### "Make More Objects"

This brings up a general object-oriented design principle that I first heard spoken by Grady Booch: "If the design is too complicated, make more objects." This is simultaneously counterintuitive and ludicrously simple, and yet it's the most useful guideline I've found. (You might observe that "making more objects" is often equivalent to "add another level of indirection.") In general, if you find a place with messy code, consider what sort of class would clean that up. Often the side effect of cleaning up the code will be a system that has better structure and is more flexible.

Consider first the place where **Trash** objects are created, which is a **switch** statement inside **main( )**:

```
# PatternRefactoring/clip1.py
    for(int i = 0 i < 30 i++)
        switch((int)(Math.random() * 3)):
            case 0 :
                bin.add(new
                  Aluminum(Math.random() * 100))
                break
            case 1 :
                bin.add(new
                  Paper(Math.random() * 100))
                break
            case 2 :
                bin.add(new
                  Glass(Math.random() * 100))
```

This is definitely messy, and also a place where you must change code whenever a new type is added. If new types are commonly added, a better solution is a single method that takes all of the necessary information and produces a reference to an object of the correct type, already upcast to a trash object. In *Design Patterns* this is broadly referred to as a *creational pattern* (of which there are several). The specific pattern that will be

applied here is a variant of the *Factory Method*. Here, the factory method is a **static** member of **Trash**, but more commonly it is a method that is overridden in the derived class.

The idea of the factory method is that you pass it the essential information it needs to know to create your object, then stand back and wait for the reference (already upcast to the base type) to pop out as the return value. From then on, you treat the object polymorphically. Thus, you never even need to know the exact type of object that's created. In fact, the factory method hides it from you to prevent accidental misuse. If you want to use the object without polymorphism, you must explicitly use RTTI and casting.

But there's a little problem, especially when you use the more complicated approach (not shown here) of making the factory method in the base class and overriding it in the derived classes. What if the information required in the derived class requires more or different arguments? "Creating more objects" solves this problem. To implement the factory method, the **Trash** class gets a new method called **factory**. To hide the creational data, there's a new class called **Messenger** that carries all of the necessary information for the **factory** method to create the appropriate **Trash** object (we've started referring to *Messenger* as a design pattern, but it's simple enough that you may not choose to elevate it to that status). Here's a simple implementation of **Messenger**:

```
# PatternRefactoring/clip2.py
class Messenger:
    # Must change this to add another type:
    MAX_NUM = 4
    def __init__(self, typeNum, val):
        self.type = typeNum % MAX_NUM
        self.data = val
```

A **Messenger** object's only job is to hold information for the **factory( )** method. Now, if there's a situation in which **factory( )** needs more or different information to create a new type of **Trash** object, the **factory( )** interface doesn't need to be changed. The **Messenger** class can be changed by adding new data and new constructors, or in the more typical object-oriented fashion of subclassing.

The **factory( )** method for this simple example looks like this:

```
# PatternRefactoring/clip3.py
    def factory(messenger):
        switch(messenger.type):
            default: # To quiet the compiler
            case 0:
                return Aluminum(messenger.data)
            case 1:
                return Paper(messenger.data)
            case 2:
                return Glass(messenger.data)
            # Two lines here:
            case 3:
                return Cardboard(messenger.data)
```

Here, the determination of the exact type of object is simple, but you can imagine a more complicated system in which **factory( )** uses an elaborate algorithm. The point is that it's now hidden away in one place, and you know to come to this place when you add new types.

The creation of new objects is now much simpler in **main( )**:

```
# PatternRefactoring/clip4.py
    for(int i = 0 i < 30 i++)
        bin.add(
          Trash.factory(
            Messenger(
```

```
                (int)(Math.random() * Messenger.MAX_NUM),
                Math.random() * 100)))
```

A **Messenger** object is created to pass the data into **factory( )**, which in turn produces some kind of **Trash** object on the heap and returns the reference that's added to the **ArrayList bin**. Of course, if you change the quantity and type of argument, this statement will still need to be modified, but that can be eliminated if the creation of the **Messenger** object is automated. For example, an **ArrayList** of arguments can be passed into the constructor of a **Messenger** object (or directly into a **factory( )** call, for that matter). This requires that the arguments be parsed and checked at run time, but it does provide the greatest flexibility.

You can see from this code what "vector of change" problem the factory is responsible for solving: if you add new types to the system (the change), the only code that must be modified is within the factory, so the factory isolates the effect of that change.

## A Pattern for Prototyping Creation

A problem with the design above is that it still requires a central location where all the types of the objects must be known: inside the **factory( )** method. If new types are regularly being added to the system, the **factory( )** method must be changed for each new type. When you discover something like this, it is useful to try to go one step further and move *all* of the information about the type-including its creation-into the class representing that type. This way, the only thing you need to do to add a new type to the system is to inherit a single class.

To move the information concerning type creation into each specific type of **Trash**, the "prototype" pattern (from the *Design Patterns* book) will be used. The general idea is that you have a master sequence of objects, one of each type you're interested in making. The objects in this sequence are used *only* for making new objects, using an operation that's not unlike the **clone( )** scheme built into Java's root class **Object**. In this case, we'll name the cloning method **tClone( )**. When you're ready to make a new object, presumably you have some sort of information that establishes the type of object you want to create, then you move through the master sequence comparing your information with whatever appropriate information is in the prototype objects in the master sequence. When you find one that matches your needs, you clone it.

In this scheme there is no hard-coded information for creation. Each object knows how to expose appropriate information and how to clone itself. Thus, the **factory( )** method doesn't need to be changed when a new type is added to the system.

One approach to the problem of prototyping is to add a number of methods to support the creation of new objects. However, in Java 1.1 there's already support for creating new objects if you have a reference to the **Class** object. With Java 1.1 *reflection* (introduced in *Thinking in Java*) you can call a constructor even if you have only a reference to the **Class** object. This is the perfect solution for the prototyping problem.

The list of prototypes will be represented indirectly by a list of references to all the **Class** objects you want to create. In addition, if the prototyping fails, the **factory( )** method will assume that it's because a particular **Class** object wasn't in the list, and it will attempt to load it. By loading the prototypes dynamically like this, the **Trash** class doesn't need to know what types it is working with, so it doesn't need any modifications when you add new types. This allows it to be easily reused throughout the rest of the chapter:

```
# PatternRefactoring/trash/Trash.py
# Base class for Trash recycling examples.

class Trash:
    def __init__(self, wt): self.weight = wt
    def getValue(self): pass
    def getWeight(self): return weight
    # Sums the value of Trash given an
```

```
        # Iterator to any container of Trash:
        def sumValue(self, it):
            val = 0.0f
            while(it.hasNext()):
                # One kind of RTTI:
                # A dynamically-checked cast
                Trash t = (Trash)it.next()
                val += t.getWeight() * t.getValue()
                print (
                  "weight of " +
                  # Using RTTI to get type
                  # information about the class:
                  t.getClass().getName() +
                  " = " + t.getWeight())

            print("Total value = " + val)

        # Remainder of class provides
        # support for prototyping:
        trashTypes = ArrayList()
        def factory(self, info):
            for i in trashTypes:
                # Somehow determine the type
                # to create, and create one:
                tc = trashTypes.get(i)
                if (tc.getName().index(info.id) != -1):
                    try:
                        # Get the dynamic constructor method
                        # that takes a double argument:
                        ctor = tc.getConstructor(type(double))
                        # Call the constructor
                        # to create a object:
                        return (Trash)ctor.newInstance()
                    except ex:
                        ex.printStackTrace(System.err)
                        raise "Cannot Create Trash"

            # Class was not in the list. Try to load it,
            # but it must be in your class path!
            try:
                print("Loading " + info.id)
                trashTypes.add(Class.forName(info.id))
            except e:
                e.printStackTrace(System.err)
                raise "Prototype not found"

            # Loaded successfully.
            # Recursive call should work:
            return factory(info)

        class Messenger:
            def __init__(self, name, val):
                self.id = name
                self.data = val

The basic **Trash** class and **sumValue( )** remain as before. The rest of the
class supports the prototyping pattern. You first see two inner classes (which
are made **static**, so they are inner classes only for code organization
```

```
purposes) describing exceptions that can occur. This is followed by an
**ArrayList** called **trashTypes**, which is used to hold the **Class**
references.
```

In **Trash.factory( )**, the **String** inside the **Messenger** object **id** (a different version of the **Messenger** class than that of the prior discussion) contains the type name of the **Trash** to be created; this **String** is compared to the **Class** names in the list. If there's a match, then that's the object to create. Of course, there are many ways to determine what object you want to make. This one is used so that information read in from a file can be turned into objects.

Once you've discovered which kind of **Trash** to create, then the reflection methods come into play. The **getConstructor( )** method takes an argument that's an array of **Class** references. This array represents the arguments, in their proper order, for the constructor that you're looking for. Here, the array is dynamically created using the Java 1.1 array-creation syntax:

```
Class[]:double.class
```

This code assumes that every **Trash** type has a constructor that takes a **double** (and notice that **double.class** is distinct from **Double.class**). It's also possible, for a more flexible solution, to call **getConstructors( )**, which returns an array of the possible constructors.

What comes back from **getConstructor( )** is a reference to a **Constructor** object (part of **java.lang.reflect**). You call the constructor dynamically with the method **newInstance( )**, which takes an array of **Object** containing the actual arguments. This array is again created using the Java 1.1 syntax:

```
Object[]{Double(Messenger.data)
```

In this case, however, the **double** must be placed inside a wrapper class so that it can be part of this array of objects. The process of calling **newInstance( )** extracts the **double**, but you can see it is a bit confusing-an argument might be a **double** or a **Double**, but when you make the call you must always pass in a **Double**. Fortunately, this issue exists only for the primitive types.

Once you understand how to do it, the process of creating a new object given only a **Class** reference is remarkably simple. Reflection also allows you to call methods in this same dynamic fashion.

Of course, the appropriate **Class** reference might not be in the **trashTypes** list. In this case, the **return** in the inner loop is never executed and you'll drop out at the end. Here, the program tries to rectify the situation by loading the **Class** object dynamically and adding it to the **trashTypes** list. If it still can't be found something is really wrong, but if the load is successful then the **factory** method is called recursively to try again.

As you'll see, the beauty of this design is that this code doesn't need to be changed, regardless of the different situations it will be used in (assuming that all **Trash** subclasses contain a constructor that takes a single **double** argument).

## Trash Subclasses

To fit into the prototyping scheme, the only thing that's required of each new subclass of **Trash** is that it contain a constructor that takes a **double** argument. Java reflection handles everything else.

Here are the different types of **Trash**, each in their own file but part of the **Trash** package (again, to facilitate reuse within the chapter):

```
# PatternRefactoring/trash/Aluminum.py
# The Aluminum class with prototyping.

class Aluminum(Trash):
```

```
    val = 1.67f
    def __init__(self, wt): Trash.__init__(wt)
    def getValue(self): return val
    def setValue(self, newVal):
        self.val = newVal::


# PatternRefactoring/trash/Paper.py
# The Paper class with prototyping.

class Paper(Trash):
    val = 0.10f
    def __init__(self, wt): Trash.__init__(wt)
    def getValue(self): return self.val
    def setValue(self, newVal):
        self.val = newVal::


# PatternRefactoring/trash/Glass.py
# The Glass class with prototyping.

class Glass(Trash):
    val = 0.23f
    def __init__(self, wt): Trash.__init__(wt)
    def getValue(self): return self.val
    def setValue(self, newVal):
        self.val = newVal
```

And here's a new type of **Trash**:

```
# PatternRefactoring/trash/Cardboard.py
# The Cardboard class with prototyping.

class Cardboard(Trash):
    val = 0.23f
    def __init__(self, wt): Trash.__init__(wt)
    def getValue(self): return self.val
    def setValue(self, newVal):
        self.val = newVal
```

You can see that, other than the constructor, there's nothing special about any of these classes.

## Parsing Trash from an External File

The information about **Trash** objects will be read from an outside file. The file has all of the necessary information about each piece of trash on a single line in the form **Trash:weight**, such as:

```
# PatternRefactoring/trash/Trash.dat
patternRefactoring.trash.Glass:54
patternRefactoring.trash.Paper:22
patternRefactoring.trash.Paper:11
patternRefactoring.trash.Glass:17
patternRefactoring.trash.Aluminum:89
patternRefactoring.trash.Paper:88
patternRefactoring.trash.Aluminum:76
patternRefactoring.trash.Cardboard:96
patternRefactoring.trash.Aluminum:25
```

```
patternRefactoring.trash.Aluminum:34
patternRefactoring.trash.Glass:11
patternRefactoring.trash.Glass:68
patternRefactoring.trash.Glass:43
patternRefactoring.trash.Aluminum:27
patternRefactoring.trash.Cardboard:44
patternRefactoring.trash.Aluminum:18
patternRefactoring.trash.Paper:91
patternRefactoring.trash.Glass:63
patternRefactoring.trash.Glass:50
patternRefactoring.trash.Glass:80
patternRefactoring.trash.Aluminum:81
patternRefactoring.trash.Cardboard:12
patternRefactoring.trash.Glass:12
patternRefactoring.trash.Glass:54
patternRefactoring.trash.Aluminum:36
patternRefactoring.trash.Aluminum:93
patternRefactoring.trash.Glass:93
patternRefactoring.trash.Paper:80
patternRefactoring.trash.Glass:36
patternRefactoring.trash.Glass:12
patternRefactoring.trash.Glass:60
patternRefactoring.trash.Paper:66
patternRefactoring.trash.Aluminum:36
patternRefactoring.trash.Cardboard:22
```

Note that the class path must be included when giving the class names, otherwise the class will not be found.

This file is read using the previously-defined **StringList** tool, and each line is picked aparat using the **String** method **indexOf( )** to produce the index of the ':'. This is first used with the **String** method **substring( )** to extract the name of the trash type, and next to get the weight that is turned into a **double** with the **static Double.valueOf( )** method. The **trim( )** method removes white space at both ends of a string.

The **Trash** parser is placed in a separate file since it will be reused throughout this chapter:

```
# PatternRefactoring/trash/ParseTrash.py
# Parse file contents into Trash objects,
# placing each into a Fillable holder.

class ParseTrash:
    def fillBin(String filename, Fillable bin):
        for line in open(filename).readlines():
            String type = line.substring(0,
              line.index(':')).strip()
            weight = Double.valueOf(
              line.substring(line.index(':') + 1)
                .strip()).doubleValue()
            bin.addTrash(
              Trash.factory(
                Trash.Messenger(type, weight)))

    # Special case to handle Collection:
    def fillBin(String filename, Bin):
        fillBin(filename, FillableCollection(bin))
```

In **RecycleA.py**, an **ArrayList** was used to hold the **Trash** objects. However, other types of containers can be used as well. To allow for this, the first version of **fillBin( )** takes a reference to a **Fillable**, which is simply

an **interface** that supports a method called **addTrash( )**:

```
# PatternRefactoring/trash/Fillable.py
# Any object that can be filled with Trash.

class Fillable:
    def addTrash(self, Trash t)
```

Anything that supports this interface can be used with **fillBin**. Of course, **Collection** doesn't implement **Fillable**, so it won't work. Since **Collection** is used in most of the examples, it makes sense to add a second overloaded **fillBin( )** method that takes a **Collection**. Any **Collection** can then be used as a **Fillable** object using an adapter class:

```
# PatternRefactoring/trash/FillableCollection.py
# Adapter that makes a Collection Fillable.

class FillableCollection(Fillable):
    def __init__(self, cc):
        self.c = cc

    def addTrash(self, t):
        self.c.add(t)
```

You can see that the only job of this class is to connect **Fillable**'s **addTrash( )** method to **Collection**'s **add( )**. With this class in hand, the overloaded **fillBin( )** method can be used with a **Collection** in **ParseTrash.py**:

```
def fillBin(filename, bin):
    fillBin(filename, FillableCollection(bin))
```

This approach works for any container class that's used frequently. Alternatively, the container class can provide its own adapter that implements **Fillable**. (You'll see this later, in **DynaTrash.py**.)

## Recycling with Prototyping

Now you can see the revised version of **RecycleA.py** using the prototyping technique:

```
# PatternRefactoring/recycleap/RecycleAP.py
# Recycling with RTTI and Prototypes.

class RecycleAP(UnitTest):
    Collection
        bin = ArrayList(),
        glassBin = ArrayList(),
        paperBin = ArrayList(),
        alBin = ArrayList()
    def __init__(self):
        # Fill up the Trash bin:
        ParseTrash.fillBin(
          "../trash/Trash.dat", bin)

    def test(self):
        Iterator sorter = bin.iterator()
        # Sort the Trash:
        while(sorter.hasNext()):
            Object t = sorter.next()
            # RTTI to show class membership:
            if(t instanceof Aluminum)
```

```
                alBin.add(t)
            if(t instanceof Paper)
                paperBin.add(t)
            if(t instanceof Glass)
                glassBin.add(t)

        Trash.sumValue(alBin.iterator())
        Trash.sumValue(paperBin.iterator())
        Trash.sumValue(glassBin.iterator())
        Trash.sumValue(bin.iterator())

    def main(self, String args[]):
        RecycleAP().test()
```

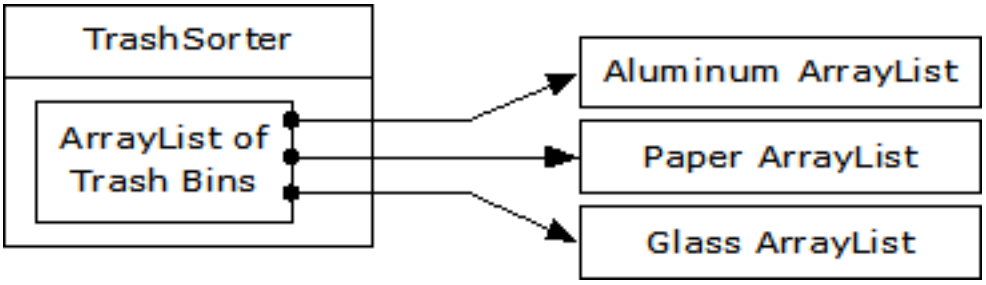
All of the **Trash** objects, as well as the **ParseTrash** and support classes, are now part of the package **pattern-Refactoring.trash**, so they are simply imported.

The process of opening the data file containing **Trash** descriptions and the parsing of that file have been wrapped into the **static** method **ParseTrash.fillBin( )**, so now it's no longer a part of our design focus. You will see that throughout the rest of the chapter, no matter what new classes are added, **ParseTrash.fillBin( )** will continue to work without change, which indicates a good design.

In terms of object creation, this design does indeed severely localize the changes you need to make to add a new type to the system. However, there's a significant problem in the use of RTTI that shows up clearly here. The program seems to run fine, and yet it never detects any cardboard, even though there is cardboard in the list! This happens *because* of the use of RTTI, which looks for only the types that you tell it to look for. The clue that RTTI is being misused is that *every type in the system* is being tested, rather than a single type or subset of types. As you will see later, there are ways to use polymorphism instead when you're testing for every type. But if you use RTTI a lot in this fashion, and you add a new type to your system, you can easily forget to make the necessary changes in your program and produce a difficult-to- find bug. So it's worth trying to eliminate RTTI in this case, not just for aesthetic reasons-it produces more maintainable code.

## Abstracting Usage

With creation out of the way, it's time to tackle the remainder of the design: where the classes are used. Since it's the act of sorting into bins that's particularly ugly and exposed, why not take that process and hide it inside a class? This is the principle of "If you must do something ugly, at least localize the ugliness inside a class." It looks like this:



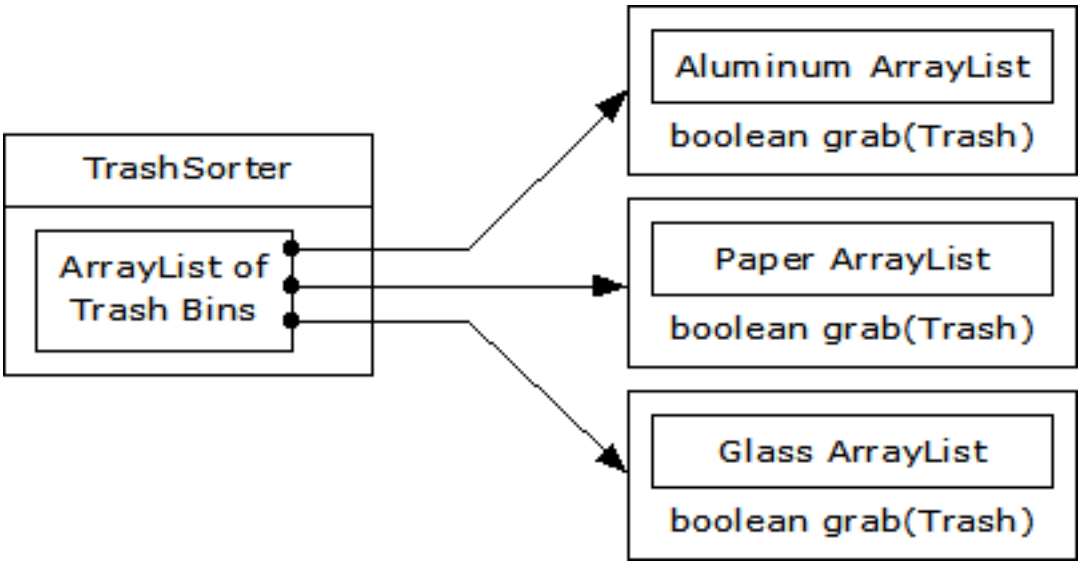

The **TrashSorter** object initialization must now be changed whenever a new type of **Trash** is added to the model. You could imagine that the **TrashSorter** class might look something like this:

```
class TrashSorter(ArrayList):
  def sort(self, Trash t): /* ... */
```

That is, **TrashSorter** is an **ArrayList** of references to **ArrayList**s of **Trash** references, and with **add( )** you can install another one, like so:

```
TrashSorter ts = TrashSorter()
ts.add(ArrayList())
```

Now, however, **sort( )** becomes a problem. How does the statically-coded method deal with the fact that a new type has been added? To solve this, the type information must be removed from **sort( )** so that all it needs to do is call a generic method that takes care of the details of type. This, of course, is another way to describe a dynamically-bound method. So **sort( )** will simply move through the sequence and call a dynamically-bound method for each **ArrayList**. Since the job of this method is to grab the pieces of trash it is interested in, it's called **grab(Trash)**. The structure now looks like:




**TrashSorter** needs to call each **grab( )** method and get a different result depending on what type of **Trash** the current **ArrayList** is holding. That is, each **ArrayList** must be aware of the type it holds. The classic approach to this problem is to create a base **Trash** bin class and inherit a new derived class for each different type you want to hold. If Java had a parameterized type mechanism that would probably be the most straightforward approach. But rather than hand-coding all the classes that such a mechanism should be building for us, further observation can produce a better approach.

A basic OOP design principle is "Use data members for variation in state, use polymorphism for variation in behavior." Your first thought might be that the **grab( )** method certainly behaves differently for an **ArrayList** that holds **Paper** than for one that holds **Glass**. But what it does is strictly dependent on the type, and nothing else. This could be interpreted as a different state, and since Java has a class to represent type (**Class**) this can be used to determine the type of **Trash** a particular **Tbin** will hold.

The constructor for this **Tbin** requires that you hand it the **Class** of your choice. This tells the **ArrayList** what type it is supposed to hold. Then the **grab( )** method uses **Class BinType** and RTTI to see if the **Trash** object you've handed it matches the type it's supposed to grab.

Here is the new version of the program:

```
# PatternRefactoring/recycleb/RecycleB.py
# Containers that grab objects of interest.

# A container that admits only the right type
# of Trash (established in the constructor):
class Tbin:
    def __init__(self, binType):
```

```
        self.list = ArrayList()
        self.type = binType
    def grab(self, t):
        # Comparing class types:
        if(t.getClass().equals(self.type)):
            self.list.add(t)
            return True # Object grabbed

        return False # Object not grabbed

    def iterator(self):
        return self.list.iterator()

class TbinList(ArrayList):
    def sort(self, Trash t):
        Iterator e = iterator() # Iterate over self
        while(e.hasNext())
            if(((Tbin)e.next()).grab(t)) return
        # Need a Tbin for this type:
        add(Tbin(t.getClass()))
        sort(t) # Recursive call

class RecycleB(UnitTest):
    Bin = ArrayList()
    TbinList trashBins = TbinList()
    def __init__(self):
        ParseTrash.fillBin("../trash/Trash.dat",bin)

    def test(self):
        Iterator it = bin.iterator()
        while(it.hasNext())
            trashBins.sort((Trash)it.next())
        Iterator e = trashBins.iterator()
        while(e.hasNext()):
            Tbin b = (Tbin)e.next()
            Trash.sumValue(b.iterator())

        Trash.sumValue(bin.iterator())

    def main(self, String args[]):
        RecycleB().test()
```

**Tbin** contains a **Class** reference **type** which establishes in the constructor what what type it should grab. The **grab()** method checks this type against the object you pass it. Note that in this design, **grab()** only accepts **Trash** objects so you get compile-time type checking on the base type, but you could also just accept **Object** and it would still work.

**TTbinList** holds a set of **Tbin** references, so that **sort( )** can iterate through the **Tbin**s when it's looking for a match for the **Trash** object you've handed it. If it doesn't find a match, it creates a new **Tbin** for the type that hasn't been found, and makes a recursive call to itself - the next time around, the new bin will be found.

Notice the genericity of this code: it doesn't change at all if new types are added. If the bulk of your code doesn't need changing when a new type is added (or some other change occurs) then you have an easily extensible system.

## Multiple Dispatching

The above design is certainly satisfactory. Adding new types to the system consists of adding or modifying distinct classes without causing code changes to be propagated throughout the system. In addition, RTTI is not "misused" as it was in **RecycleA.py**. However, it's possible to go one step further and take a purist viewpoint about RTTI and say that it should be eliminated altogether from the operation of sorting the trash into bins.

To accomplish this, you must first take the perspective that all type-dependent activities-such as detecting the type of a piece of trash and putting it into the appropriate bin-should be controlled through polymorphism and dynamic binding.

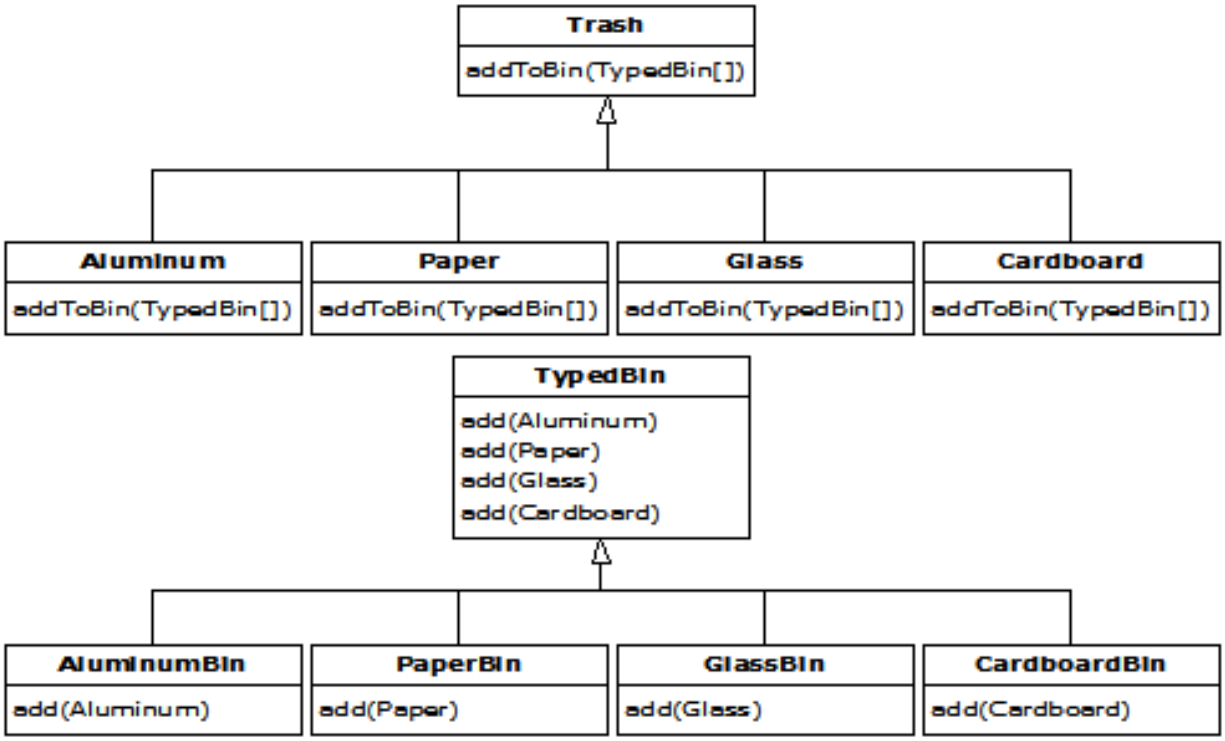
The previous examples first sorted by type, then acted on sequences of elements that were all of a particular type. But whenever you find yourself picking out particular types, stop and think. The whole idea of polymorphism (dynamically- bound method calls) is to handle type-specific information for you. So why are you hunting for types?

The answer is something you probably don't think about: Python performs only single dispatching. That is, if you are performing an operation on more than one object whose type is unknown, Python will invoke the dynamic binding mechanism on only one of those types. This doesn't solve the problem, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

The solution is called *multiple dispatching*, which means setting up a configuration such that a single method call produces more than one dynamic method call and thus determines more than one type in the process. To get this effect, you need to work with more than one type hierarchy: you'll need a type hierarchy for each dispatch. The following example works with two hierarchies: the existing **Trash** family and a hierarchy of the types of trash bins that the trash will be placed into. This second hierarchy isn't always obvious and in this case it needed to be created in order to produce multiple dispatching (in this case there will be only two dispatches, which is referred to as *double dispatching*).

### Implementing the Double Dispatch

Remember that polymorphism can occur only via method calls, so if you want double dispatching to occur, there must be two method calls: one used to determine the type within each hierarchy. In the Trash hierarchy there will be a new method called addToBin( ), which takes an argument of an array of TypedBin. It uses this array to step through and try to add itself to the appropriate bin, and this is where you'll see the double dispatch.




The new hierarchy is TypedBin, and it contains its own method called add( ) that is also used polymorphically. But here's an additional twist: add( ) is overloaded to take arguments of the different types of trash. So an essential part of the double dispatching scheme also involves overloading.

Redesigning the program produces a dilemma: it's now necessary for the base class **Trash** to contain an **addToBin( )** method. One approach is to copy all of the code and change the base class. Another approach, which you can take when you don't have control of the source code, is to put the **addToBin( )** method into an **interface**, leave **Trash** alone, and inherit new specific types of **Aluminum**, **Paper**, **Glass**, and **Cardboard**. This is the approach that will be taken here.

Most of the classes in this design must be **public**, so they are placed in their own files. Here's the interface:

```
# PatternRefactoring/doubledispatch/TypedBinMember.py
# An class for adding the double
# dispatching method to the trash hierarchy
# without modifying the original hierarchy.

class TypedBinMember:
    # The method:
    boolean addToBin(TypedBin[] tb)
```

In each particular subtype of **Aluminum**, **Paper**, **Glass,** and **Cardboard**, the **addToBin( )** method in the **interface TypedBinMember** is implemented, but it *looks* like the code is exactly the same in each case:

```
# PatternRefactoring/doubledispatch/DDAluminum.py
# Aluminum for double dispatching.

class DDAluminum(Aluminum, TypedBinMember):
    def __init__(self, wt): Aluminum.__init__(wt)
    def addToBin(self, TypedBin[] tb):
        for(int i = 0 i < tb.length i++)
            if(tb[i].add(self)):
```

```
                return True
        return False::


# PatternRefactoring/doubledispatch/DDPaper.py
# Paper for double dispatching.

class DDPaper(Paper, TypedBinMember):
    def __init__(self, wt): Paper.__init__(wt)
    def addToBin(self, TypedBin[] tb):
        for(int i = 0 i < tb.length i++)
            if(tb[i].add(self))
                return True
        return False::


# PatternRefactoring/doubledispatch/DDGlass.py
# Glass for double dispatching.

class DDGlass(Glass, TypedBinMember):
    def __init__(self, wt): Glass.__init__(wt)
    def addToBin(self, TypedBin[] tb):
        for(int i = 0 i < tb.length i++)
            if(tb[i].add(self))
                return True
        return False::


# PatternRefactoring/doubledispatch/DDCardboard.py
# Cardboard for double dispatching.

class DDCardboard(Cardboard, TypedBinMember):
    def __init__(self, wt):
        Cardboard.__init__(wt)
    def addToBin(self, TypedBin[] tb):
        for(int i = 0 i < tb.length i++)
            if(tb[i].add(self))
                return True
        return False
```

The code in each **addToBin( )** calls **add( )** for each **TypedBin** object in the array. But notice the argument: **this**. The type of **this** is different for each subclass of **Trash**, so the code is different. (Although this code will benefit if a parameterized type mechanism is ever added to Java.) So this is the first part of the double dispatch, because once you're inside this method you know you're **Aluminum**, or **Paper**, etc. During the call to **add( )**, this information is passed via the type of **this**. The compiler resolves the call to the proper overloaded version of **add( )**. But since **tb[i]** produces a reference to the base type **TypedBin**, this call will end up calling a different method depending on the type of **TypedBin** that's currently selected. That is the second dispatch.

Here's the base class for **TypedBin**:

```
# PatternRefactoring/doubledispatch/TypedBin.py
# A container for the second dispatch.

class TypedBin:
    c = ArrayList()
    def addIt(self, Trash t):
        c.add(t)
```

```
        return True

    def iterator(self):
        return c.iterator()

    def add(self, DDAluminum a):
        return False

    def add(self, DDPaper a):
        return False

    def add(self, DDGlass a):
        return False

    def add(self, DDCardboard a):
        return False
```

You can see that the overloaded **add( )** methods all return **false**. If the method is not overloaded in a derived class, it will continue to return **false**, and the caller (**addToBin( )**, in this case) will assume that the current **Trash** object has not been added successfully to a container, and continue searching for the right container.

In each of the subclasses of **TypedBin**, only one overloaded method is overridden, according to the type of bin that's being created. For example, **CardboardBin** overrides **add(DDCardboard)**. The overridden method adds the trash object to its container and returns **true**, while all the rest of the **add( )** methods in **CardboardBin** continue to return **false**, since they haven't been overridden. This is another case in which a parameterized type mechanism in Java would allow automatic generation of code. (With C++ **template**s, you wouldn't have to explicitly write the subclasses or place the **addToBin( )** method in **Trash**.)

Since for this example the trash types have been customized and placed in a different directory, you'll need a different trash data file to make it work. Here's a possible **DDTrash.dat**:

```
# PatternRefactoring/doubledispatch/DDTrash.dat
DDGlass:54
DDPaper:22
DDPaper:11
DDGlass:17
DDAluminum:89
DDPaper:88
DDAluminum:76
DDCardboard:96
DDAluminum:25
DDAluminum:34
DDGlass:11
DDGlass:68
DDGlass:43
DDAluminum:27
DDCardboard:44
DDAluminum:18
DDPaper:91
DDGlass:63
DDGlass:50
DDGlass:80
DDAluminum:81
DDCardboard:12
DDGlass:12
DDGlass:54
DDAluminum:36
DDAluminum:93
```

```
DDGlass:93
DDPaper:80
DDGlass:36
DDGlass:12
DDGlass:60
DDPaper:66
DDAluminum:36
DDCardboard:22
```

Here's the rest of the program:

```
# PatternRefactoring/doubledispatch/DoubleDispatch.py
# Using multiple dispatching to handle more
# than one unknown type during a method call.

class AluminumBin(TypedBin):
    def add(self, DDAluminum a):
        return addIt(a)

class PaperBin(TypedBin):
    def add(self, DDPaper a):
        return addIt(a)

class GlassBin(TypedBin):
    def add(self, DDGlass a):
        return addIt(a)

class CardboardBin(TypedBin):
    def add(self, DDCardboard a):
        return addIt(a)

class TrashBinSet:
    binSet = [
      AluminumBin(),
      PaperBin(),
      GlassBin(),
      CardboardBin()
    ]

    def sortIntoBins(self, bin):
        Iterator e = bin.iterator()
        while(e.hasNext()):
            TypedBinMember t =
                (TypedBinMember)e.next()
            if(!t.addToBin(binSet))
                System.err.println("Couldn't add " + t)

    def binSet(): return binSet

class DoubleDispatch(UnitTest):
    Bin = ArrayList()
    TrashBinSet bins = TrashBinSet()
    def __init__(self):
        # ParseTrash still works, without changes:
        ParseTrash.fillBin("DDTrash.dat", bin)

    def test(self):
        # Sort from the master bin into
```

```
        # the individually-typed bins:
        bins.sortIntoBins(bin)
        TypedBin[] tb = bins.binSet()
        # Perform sumValue for each bin...
        for(int i = 0 i < tb.length i++)
            Trash.sumValue(tb[i].c.iterator())
        # ... and for the master bin
        Trash.sumValue(bin.iterator())

    def main(self, String args[]):
        DoubleDispatch().test()
```

**TrashBinSet** encapsulates all of the different types of **TypedBin**s, along with the **sortIntoBins( )** method, which is where all the double dispatching takes place. You can see that once the structure is set up, sorting into the various **TypedBin**s is remarkably easy. In addition, the efficiency of two dynamic method calls is probably better than any other way you could sort.

Notice the ease of use of this system in **main( )**, as well as the complete independence of any specific type information within **main( )**. All other methods that talk only to the **Trash** base-class interface will be equally invulnerable to changes in **Trash** types.

The changes necessary to add a new type are relatively isolated: you modify **TypedBin**, inherit the new type of **Trash** with its **addToBin( )** method, then inherit a new **TypedBin** (this is really just a copy and simple edit), and finally add a new type into the aggregate initialization for **TrashBinSet**.

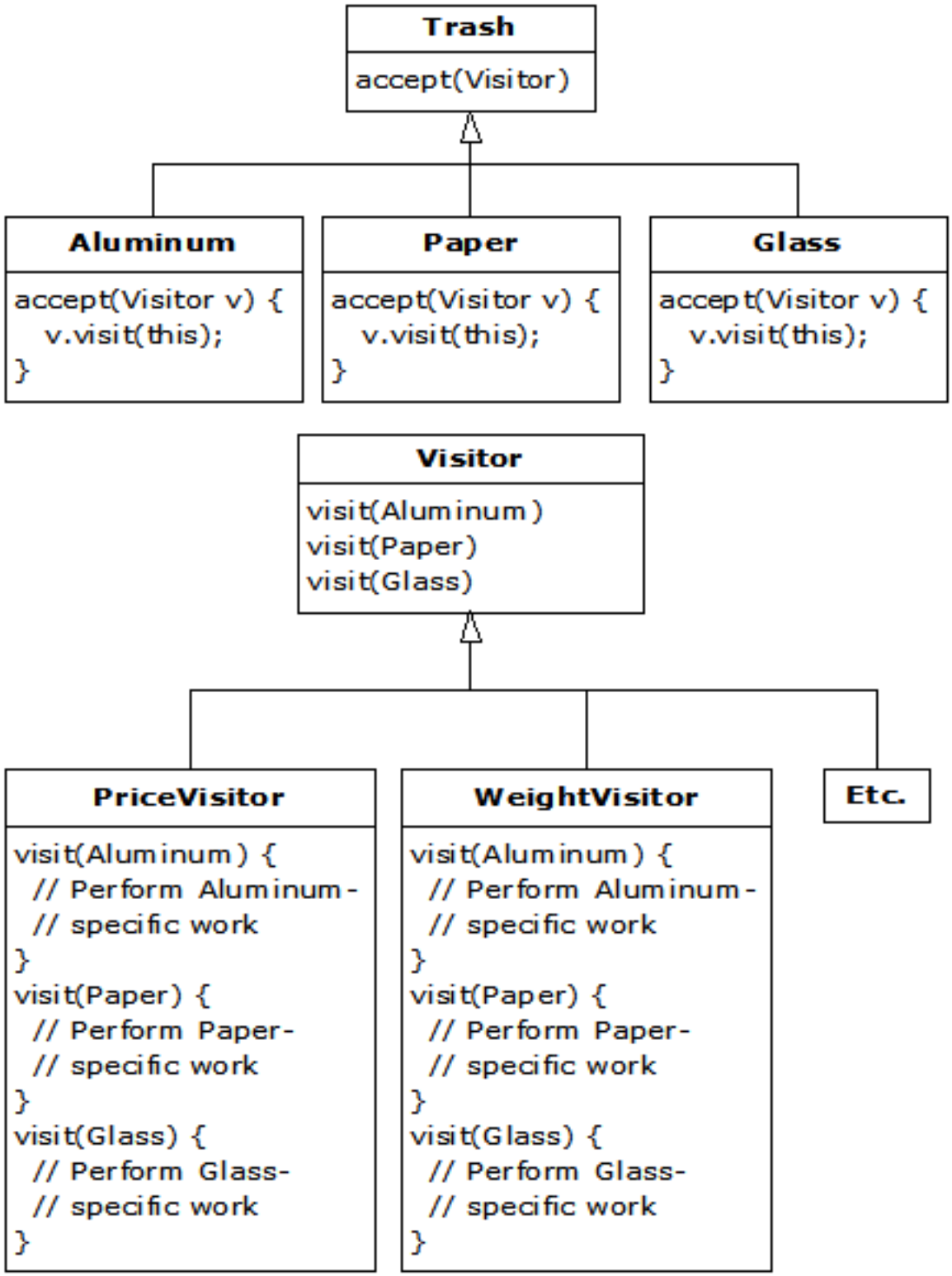
## The *Visitor* Pattern

Now consider applying a design pattern that has an entirely different goal to the trash sorting problem.

For this pattern, we are no longer concerned with optimizing the addition of new types of **Trash** to the system. Indeed, this pattern makes adding a new type of **Trash** *more* complicated. The assumption is that you have a primary class hierarchy that is fixed; perhaps it's from another vendor and you can't make changes to that hierarchy. However, you'd like to add new polymorphic methods to that hierarchy, which means that normally you'd have to add something to the base class interface. So the dilemma is that you need to add methods to the base class, but you can't touch the base class. How do you get around this?

The design pattern that solves this kind of problem is called a "visitor" (the final one in the *Design Patterns* book), and it builds on the double dispatching scheme shown in the last section.

The visitor pattern allows you to extend the interface of the primary type by creating a separate class hierarchy of type **Visitor** to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound method. It looks like this:




Now, if **v** is a **Visitable** reference to an **Aluminum** object, the code:

```
PriceVisitor pv = PriceVisitor()
v.accept(pv)
```

uses double dispatching to cause two polymorphic method calls: the first one to select **Aluminum**'s version of **accept( )**, and the second one within **accept( )** when the specific version of **visit( )** is called dynamically using the base-class **Visitor** reference **v**.

This configuration means that new functionality can be added to the system in the form of new subclasses of **Visitor**. The **Trash** hierarchy doesn't need to be touched. This is the prime benefit of the visitor pattern: you can add new polymorphic functionality to a class hierarchy without touching that hierarchy (once the **accept( )** methods have been installed). Note that the benefit is helpful here but not exactly what we started out to accomplish, so at first blush you might decide that this isn't the desired solution.

But look at one thing that's been accomplished: the visitor solution avoids sorting from the master **Trash** sequence into individual typed sequences. Thus, you can leave everything in the single master sequence and simply pass through that sequence using the appropriate visitor to accomplish the goal. Although this behavior seems to be a side effect of visitor, it does give us what we want (avoiding RTTI).

The double dispatching in the visitor pattern takes care of determining both the type of **Trash** and the type of **Visitor**. In the following example, there are two implementations of **Visitor**: **PriceVisitor** to both determine and sum the price, and **WeightVisitor** to keep track of the weights.

You can see all of this implemented in the new, improved version of the recycling program.

As with **DoubleDispatch.py**, the **Trash** class is left alone and a new interface is created to add the **accept( )** method:

```
# PatternRefactoring/trashvisitor/Visitable.py
# An class to add visitor functionality
# to the Trash hierarchy without
# modifying the base class.

class Visitable:
    # The method:
    def accept(self, Visitor v)
```

Since there's nothing concrete in the **Visitor** base class, it can be created as an **interface**:

```
# PatternRefactoring/trashvisitor/Visitor.py
# The base class for visitors.

class Visitor:
    def visit(self, Aluminum a)
    def visit(self, Paper p)
    def visit(self, Glass g)
    def visit(self, Cardboard c)
```

## A Reflective Decorator

At this point, you *could* follow the same approach that was used for double dispatching and create new subtypes of **Aluminum**, **Paper**, **Glass**, and **Cardboard** that implement the **accept( )** method. For example, the new **Visitable Aluminum** would look like this:

```
# PatternRefactoring/trashvisitor/VAluminum.py
# Taking the previous approach of creating a
# specialized Aluminum for the visitor pattern.

class VAluminum(Aluminum, Visitable):
    def __init__(self, wt): Aluminum.__init__(wt)
    def accept(self, v):
        v.visit(self)
```

However, we seem to be encountering an "explosion of interfaces:" basic **Trash**, special versions for double dispatching, and now more special versions for visitor. Of course, this "explosion of interfaces" is arbitrary-one could simply put the additional methods in the **Trash** class. If we ignore that we can instead see an opportunity to use the *Decorator* pattern: it seems like it should be possible to create a *Decorator* that can be wrapped around an ordinary **Trash** object and will produce the same interface as **Trash** and add the extra **accept( )** method. In fact, it's a perfect example of the value of *Decorator*.

The double dispatch creates a problem, however. Since it relies on overloading of both **accept( )** and **visit( )**, it would seem to require specialized code for each different version of the **accept( )** method. With C++ templates, this would be fairly easy to accomplish (since templates automatically generate type-specialized code) but Python has no such mechanism-at least it does not appear to. However, reflection allows you to determine type information at run time, and it turns out to solve many problems that would seem to require templates (albeit not as simply). Here's the decorator that does the trick[1]:

```
# PatternRefactoring/trashvisitor/VisitableDecorator.py
# A decorator that adapts the generic Trash
# classes to the visitor pattern.

class VisitableDecorator(Trash, Visitable):
    def __init__(self, t):
        self.delegate = t
        try:
            self.dispatch = Visitor.class.getMethod (
              "visit", Class[]: t.getClass()
            )
        except ex:
            ex.printStackTrace()

    def getValue(self):
        return delegate.getValue()

    def getWeight(self):
        return delegate.getWeight()

    def accept(self, Visitor v):
        self.dispatch.invoke(v, delegate)
```

[[ Description of Reflection use ]]

The only other tool we need is a new type of **Fillable** adapter that automatically decorates the objects as they are being created from the original **Trash.dat** file. But this might as well be a decorator itself, decorating any kind of **Fillable**:

```
# PatternRefactoring/trashvisitor/FillableVisitor.py
# Adapter Decorator that adds the visitable
# decorator as the Trash objects are
# being created.

class FillableVisitor(Fillable):
    def __init__(self, ff): self.f = ff
    def addTrash(self, t):
        self.f.addTrash(VisitableDecorator(t))
```

Now you can wrap it around any kind of existing **Fillable**, or any new ones that haven't yet been created.

[1] Addison-Wesley, 1999.

The rest of the program creates specific **Visitor** types and sends them through a single list of **Trash** objects:

```
# PatternRefactoring/trashvisitor/TrashVisitor.py
# The "visitor" pattern with VisitableDecorators.

# Specific group of algorithms packaged
# in each implementation of Visitor:
class PriceVisitor(Visitor):
    alSum = 0.0 # Aluminum
    pSum = 0.0  # Paper
    gSum = 0.0  # Glass
    cSum = 0.0  # Cardboard
    def visit(self, al):
        v = al.getWeight() * al.getValue()
        print("value of Aluminum= " + v)
        alSum += v

    def visit(self, p):
        v = p.getWeight() * p.getValue()
        print("value of Paper= " + v)
        pSum += v

    def visit(self, g):
        v = g.getWeight() * g.getValue()
        print("value of Glass= " + v)
        gSum += v

    def visit(self, c):
        v = c.getWeight() * c.getValue()
        print("value of Cardboard = " + v)
        cSum += v

    def total(self):
        print (
          "Total Aluminum: $" + alSum +
          "\n Total Paper: $" + pSum +
          "\nTotal Glass: $" + gSum +
          "\nTotal Cardboard: $" + cSum +
          "\nTotal: $" +
            (alSum + pSum + gSum + cSum))

class WeightVisitor(Visitor):
    alSum = 0.0  # Aluminum
    pSum = 0.0  # Paper
    gSum = 0.0  # Glass
    cSum = 0.0  # Cardboard
    def visit(self, Aluminum al):
        alSum += al.getWeight()
        print ("weight of Aluminum = "
            + al.getWeight())

    def visit(self, Paper p):
        pSum += p.getWeight()
        print ("weight of Paper = "
            + p.getWeight())

    def visit(self, Glass g):
        gSum += g.getWeight()
        print ("weight of Glass = "
```

```
                + g.getWeight())

        def visit(self, Cardboard c):
            cSum += c.getWeight()
            print ("weight of Cardboard = "
                + c.getWeight())

        def total(self):
            print (
              "Total weight Aluminum: "  + alSum +
              "\nTotal weight Paper: " + pSum +
              "\nTotal weight Glass: " + gSum +
              "\nTotal weight Cardboard: " + cSum +
              "\nTotal weight: " +
                (alSum + pSum + gSum + cSum))

    class TrashVisitor(UnitTest):
        Bin = ArrayList()
        PriceVisitor pv = PriceVisitor()
        WeightVisitor wv = WeightVisitor()
        def __init__(self):
            ParseTrash.fillBin("../trash/Trash.dat",
              FillableVisitor(
                FillableCollection(bin)))

        def test(self):
            Iterator it = bin.iterator()
            while(it.hasNext()):
                Visitable v = (Visitable)it.next()
                v.accept(pv)
                v.accept(wv)

            pv.total()
            wv.total()

        def main(self, String args[]):
            TrashVisitor().test()

In **Test( )**, note how visitability is added by simply creating a different
kind of bin using the decorator. Also notice that the **FillableCollection**
adapter has the appearance of being used as a decorator (for **ArrayList**) in
this situation. However, it completely changes the interface of the
**ArrayList**, whereas the definition of *Decorator* is that the interface of
the decorated class must still be there after decoration.
```

Note that the shape of the client code (shown in the **Test** class) has changed again, from the original approaches to the problem. Now there's only a single **Trash** bin. The two **Visitor** objects are accepted into every element in the sequence, and they perform their operations. The visitors keep their own internal data to tally the total weights and prices.

Finally, there's no run time type identification other than the inevitable cast to **Trash** when pulling things out of the sequence. This, too, could be eliminated with the implementation of parameterized types in Java.

One way you can distinguish this solution from the double dispatching solution described previously is to note that, in the double dispatching solution, only one of the overloaded methods, **add( )**, was overridden when each subclass was created, while here *each* one of the overloaded **visit( )** methods is overridden in every subclass of **Visitor**.

### More Coupling?

There's a lot more code here, and there's definite coupling between the **Trash** hierarchy and the **Visitor** hierarchy. However, there's also high cohesion within the respective sets of classes: they each do only one thing (**Trash** describes Trash, while **Visitor** describes actions performed on **Trash**), which is an indicator of a good design. Of course, in this case it works well only if you're adding new **Visitor**s, but it gets in the way when you add new types of **Trash**.

Low coupling between classes and high cohesion within a class is definitely an important design goal. Applied mindlessly, though, it can prevent you from achieving a more elegant design. It seems that some classes inevitably have a certain intimacy with each other. These often occur in pairs that could perhaps be called *couplets*; for example, containers and iterators. The **Trash- Visitor** pair above appears to be another such couplet.

## RTTI Considered Harmful?

Various designs in this chapter attempt to remove RTTI, which might give you the impression that it's "considered harmful" (the condemnation used for poor, ill- fated **goto**, which was thus never put into Java). This isn't true; it is the *misuse* of RTTI that is the problem. The reason our designs removed RTTI is because the misapplication of that feature prevented extensibility, while the stated goal was to be able to add a new type to the system with as little impact on surrounding code as possible. Since RTTI is often misused by having it look for every single type in your system, it causes code to be non-extensible: when you add a new type, you have to go hunting for all the code in which RTTI is used, and if you miss any you won't get help from the compiler.

However, RTTI doesn't automatically create non-extensible code. Let's revisit the trash recycler once more. This time, a new tool will be introduced, which I call a **TypeMap**. It contains a **HashMap** that holds **ArrayList**s, but the interface is simple: you can **add( )** a new object, and you can **get( )** an **ArrayList** containing all the objects of a particular type. The keys for the contained **HashMap** are the types in the associated **ArrayList**. The beauty of this design (suggested by Larry O'Brien) is that the **TypeMap** dynamically adds a new pair whenever it encounters a new type, so whenever you add a new type to the system (even if you add the new type at run time), it adapts.

Our example will again build on the structure of the **Trash** types in **package patternRefactoring.Trash** (and the **Trash.dat** file used there can be used here without change):

```
# PatternRefactoring/dynatrash/DynaTrash.py
# Using a Map of Lists and RTTI
# to automatically sort trash into
# ArrayLists. This solution, despite the
# use of RTTI, is extensible.

# Generic TypeMap works in any situation:
class TypeMap:
    t = HashMap()
    def add(self, Object o):
        Class type = o.getClass()
        if(self.t.has_key(type))
            self.t.get(type).add(o)
        else:
            List v = ArrayList()
            v.add(o)
            t.put(type,v)

    def get(self, Class type):
```

```
        return (List)t.get(type)

    def keys(self):
        return t.keySet().iterator()

# Adapter class to allow callbacks
# from ParseTrash.fillBin():
class TypeMapAdapter(Fillable):
    TypeMap map
    def __init__(self, TypeMap tm): map = tm
    def addTrash(self, Trash t): map.add(t)

class DynaTrash(UnitTest):
    TypeMap bin = TypeMap()
    def __init__(self):
        ParseTrash.fillBin("../trash/Trash.dat",
          TypeMapAdapter(bin))

    def test(self):
        Iterator keys = bin.keys()
        while(keys.hasNext())
            Trash.sumValue(
              bin.get((Class)keys.next()).iterator())

    def main(self, String args[]):
        DynaTrash().test()
```

Although powerful, the definition for **TypeMap** is simple. It contains a **HashMap**, and the **add( )** method does most of the work. When you **add( )** a new object, the reference for the **Class** object for that type is extracted. This is used as a key to determine whether an **ArrayList** holding objects of that type is already present in the **HashMap**. If so, that **ArrayList** is extracted and the object is added to the **ArrayList**. If not, the **Class** object and a new **ArrayList** are added as a key-value pair.

You can get an **Iterator** of all the **Class** objects from **keys( )**, and use each **Class** object to fetch the corresponding **ArrayList** with **get( )**. And that's all there is to it.

The **filler( )** method is interesting because it takes advantage of the design of **ParseTrash.fillBin( )**, which doesn't just try to fill an **ArrayList** but instead anything that implements the **Fillable** interface with its **addTrash( )** method. All **filler( )** needs to do is to return a reference to an **interface** that implements **Fillable**, and then this reference can be used as an argument to **fillBin( )** like this:

ParseTrash.fillBin("Trash.dat", bin.filler())

To produce this reference, an *anonymous inner class* (described in *Thinking in Java*) is used. You never need a named class to implement **Fillable**, you just need a reference to an object of that class, thus this is an appropriate use of anonymous inner classes.

An interesting thing about this design is that even though it wasn't created to handle the sorting, **fillBin( )** is performing a sort every time it inserts a **Trash** object into **bin**.

Much of **class DynaTrash** should be familiar from the previous examples. This time, instead of placing the new **Trash** objects into a **bin** of type **ArrayList**, the **bin** is of type **TypeMap**, so when the trash is thrown into **bin** it's immediately sorted by **TypeMap**'s internal sorting mechanism. Stepping through the **TypeMap** and operating on each individual **ArrayList** becomes a simple matter.

As you can see, adding a new type to the system won't affect this code at all, and the code in **TypeMap** is completely independent. This is certainly the smallest solution to the problem, and arguably the most elegant as well. It does rely heavily on RTTI, but notice that each key-value pair in the **HashMap** is looking

for only one type. In addition, there's no way you can "forget" to add the proper code to this system when you add a new type, since there isn't any code you need to add.

## Summary

Coming up with a design such as **TrashVisitor.py** that contains a larger amount of code than the earlier designs can seem at first to be counterproductive. It pays to notice what you're trying to accomplish with various designs. Design patterns in general strive to *separate the things that change from the things that stay the same*. The "things that change" can refer to many different kinds of changes. Perhaps the change occurs because the program is placed into a new environment or because something in the current environment changes (this could be: "The user wants to add a new shape to the diagram currently on the screen"). Or, as in this case, the change could be the evolution of the code body. While previous versions of the trash sorting example emphasized the addition of new *types* of **Trash** to the system, **TrashVisitor.py** allows you to easily add new *functionality* without disturbing the **Trash** hierarchy. There's more code in **TrashVisitor.py**, but adding new functionality to **Visitor** is cheap. If this is something that happens a lot, then it's worth the extra effort and code to make it happen more easily.

The discovery of the vector of change is no trivial matter; it's not something that an analyst can usually detect before the program sees its initial design. The necessary information will probably not appear until later phases in the project: sometimes only at the design or implementation phases do you discover a deeper or more subtle need in your system. In the case of adding new types (which was the focus of most of the "recycle" examples) you might realize that you need a particular inheritance hierarchy only when you are in the maintenance phase and you begin extending the system!

One of the most important things that you'll learn by studying design patterns seems to be an about-face from what has been promoted so far in this book. That is: "OOP is all about polymorphism." This statement can produce the "two-year- old with a hammer" syndrome (everything looks like a nail). Put another way, it's hard enough to "get" polymorphism, and once you do, you try to cast all your designs into that one particular mold.

What design patterns say is that OOP isn't just about polymorphism. It's about "separating the things that change from the things that stay the same." Polymorphism is an especially important way to do this, and it turns out to be helpful if the programming language directly supports polymorphism (so you don't have to wire it in yourself, which would tend to make it prohibitively expensive). But design patterns in general show *other* ways to accomplish the basic goal, and once your eyes have been opened to this you will begin to search for more creative designs.

Since the *Design Patterns* book came out and made such an impact, people have been searching for other patterns. You can expect to see more of these appear as time goes on. Here are some sites recommended by Jim Coplien, of C++ fame (*http://www.bell-labs.com/~cope*), who is one of the main proponents of the patterns movement:

http://st-www.cs.uiuc.edu/users/patterns http://c2.com/cgi/wiki http://c2.com/ppr http://www.bell-labs.com/people/cope/Patterns/Process/index.html http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic http://www.cs.wustl.edu/~schmidt/patterns.html http://www.espinc.com/patterns/overview.html

Also note there has been a yearly conference on design patterns, called PLOP, that produces a published proceedings, the third of which came out in late 1997 (all published by Addison-Wesley).

## Exercises

1. Add a class **Plastic** to **TrashVisitor.py**.

2. Add a class **Plastic** to **DynaTrash.py**.
3. Create a decorator like **VisitableDecorator**, but for the multiple dispatching example, along with an "adapter decorator" class like the one created for **VisitableDecorator**. Build the rest of the example and show that it works.

CHAPTER 40

# Projects

**Note:** This chapter has not had any significant translation yet.

A number of more challenging projects for you to solve. [[Some of these may turn into examples in the book, and so at some point might disappear from here]]

## Rats & Mazes

First, create a *Blackboard* (cite reference) which is an object on which anyone may record information. This particular blackboard draws a maze, and is used as information comes back about the structure of a maze from the rats that are investigating it.

Now create the maze itself. Like a real maze, this object reveals very little information about itself - given a coordinate, it will tell you whether there are walls or spaces in the four directions immediately surrounding that coordinate, but no more. For starters, read the maze in from a text file but consider hunting on the internet for a maze-generating algorithm. In any event, the result should be an object that, given a maze coordinate, will report walls and spaces around that coordinate. Also, you must be able to ask it for an entry point to the maze.

Finally, create the maze-investigating **Rat** class. Each rat can communicate with both the blackboard to give the current information and the maze to request new information based on the current position of the rat. However, each time a rat reaches a decision point where the maze branches, it creates a new rat to go down each of the branches. Each rat is driven by its own thread. When a rat reaches a dead end, it terminates itself after reporting the results of its final investigation to the blackboard.

The goal is to completely map the maze, but you must also determine whether the end condition will be naturally found or whether the blackboard must be responsible for the decision.

An example implementation by Jeremy Meyer:

```
# Projects/ratsAndMazes/Maze.py
```

```
class Maze(Canvas):
    lines = [] # a line is a char array
    width = -1
    height = -1
    main(self):
        if (args.length < 1):
            print("Enter filename")
            System.exit(0)

        Maze m = Maze()
        m.load(args[0])
        Frame f = Frame()
        f.setSize(m.width*20, m.height*20)
        f.add(m)
        Rat r = Rat(m, 0, 0)
        f.setVisible(1)

    def __init__(self):
        lines = Vector()
        setBackground(Color.lightGray)

    def isEmptyXY(x, y):
        if (x < 0) x += width
        if (y < 0) y += height
        # Use mod arithmetic to bring rat in line:
        byte[] by =
            (byte[])(lines.elementAt(y%height))
        return by[x%width]==' '

    def setXY(x, y, byte newByte):
        if (x < 0) x += width
        if (y < 0) y += height
        byte[] by =
            (byte[])(lines.elementAt(y%height))
        by[x%width] = newByte
        repaint()

    def  load(String filename):
        String currentLine = null
        BufferedReader br = BufferedReader(
          FileReader(filename))
        for(currentLine = br.readLine()
            currentLine != null
            currentLine = br.readLine()) :
            lines.addElement(currentLine.getBytes())
            if(width < 0 ||
               currentLine.getBytes().length > width)
                width = currentLine.getBytes().length

        height = len(lines)
        br.close()

    def update(self, Graphics g): paint(g)
    def paint(Graphics g):
        canvasHeight = self.getBounds().height
        canvasWidth  = self.getBounds().width
        if (height < 1 || width < 1)
            return # nothing to do
```

```
        width =
            ((byte[])(lines.elementAt(0))).length
        for (int y = 0 y < len(lines) y++):
            byte[] b
            b = (byte[])(lines.elementAt(y))
            for (int x = 0 x < width x++):
                switch(b[x]):
                    case ' ': # empty part of maze
                        g.setColor(Color.lightGray)
                        g.fillRect(
                          x*(canvasWidth/width),
                          y*(canvasHeight/height),
                          canvasWidth/width,
                          canvasHeight/height)
                        break
                    case '*':     # a wall
                        g.setColor(Color.darkGray)
                        g.fillRect(
                          x*(canvasWidth/width),
                          y*(canvasHeight/height),
                          (canvasWidth/width)-1,
                          (canvasHeight/height)-1)
                        break
                    default:      # must be rat
                        g.setColor(Color.red)
                        g.fillOval(x*(canvasWidth/width),
                        y*(canvasHeight/height),
                        canvasWidth/width,
                        canvasHeight/height)
                        break::


# Projects/ratsAndMazes/Rat.py

class Rat:
    ratCount = 0
    prison = Maze()
    vertDir = 0
    horizDir = 0
    x = 0, y = 0
    myRatNo = 0
    def __init__(self, maze, xStart, yStart):
        myRatNo = ratCount++
        print ("Rat no." + myRatNo + " ready to scurry.")
        prison = maze
        x = xStart
        y = yStart
        prison.setXY(x,y, (byte)'R')

    def run(self): scurry().start()

    def scurry(self):
        # Try and maintain direction if possible.
        # Horizontal backward
        boolean ratCanMove = 1
        while(ratCanMove):
            ratCanMove = 0
            # South
```

```
            if (prison.isEmptyXY(x, y + 1)):
                vertDir = 1 horizDir = 0
                ratCanMove = 1

            # North
            if (prison.isEmptyXY(x, y - 1))
                if (ratCanMove)
                    Rat(prison, x, y-1)
                    # Rat can move already, so give
                    # this choice to the next rat.
                else:
                    vertDir = -1 horizDir = 0
                    ratCanMove = 1

            # West
            if (prison.isEmptyXY(x-1, y))
                if (ratCanMove)
                    Rat(prison, x-1, y)
                    # Rat can move already, so give
                    # this choice to the next rat.
                else:
                    vertDir = 0 horizDir = -1
                    ratCanMove = 1

            # East
            if (prison.isEmptyXY(x+1, y))
                if (ratCanMove)
                    Rat(prison, x+1, y)
                    # Rat can move already, so give
                    # this choice to the next rat.
                else:
                    vertDir = 0 horizDir = 1
                    ratCanMove = 1

            if (ratCanMove): # Move original rat.
                x += horizDir
                y += vertDir
                prison.setXY(x,y,(byte)'R')
                # If not then the rat will die.
            Thread.sleep(2000)

        print ("Rat no." + myRatNo +
          " can't move..dying..aarrgggh.")
```

The maze initialization file:

```
# Projects/ratsAndMazes/Amaze.txt
   * **       *  * **       *
 ***     * *******     * ****
     ***           ***
 *****    **********   *****
 * * * * **  ** * * * **  *
   * * *  * **  * * *  * **
 *      **      *      **
   * **   * **  * **   * **
 *** *  *** ***** *  *** **
 *       *   * *       *   *
   * ** * *     * ** * *
```

## Other Maze Resources

A discussion of algorithms to create mazes as well as Java source code to implement them:

http://www.mazeworks.com/mazegen/mazegen.htm

A discussion of algorithms for collision detection and other individual/group moving behavior for autonomous physical objects:

http://www.red3d.com/cwr/steer/

CHAPTER 41

# Indices and tables

- genindex
- search

# Index