500 Lines or Less

500 Lines or Less

Experienced programmers solve interesting problems

Edited by Michael DiBernardo

500 Lines or Less
Edited by Michael DiBernardo

The full text of this book is available online at `http://www.aosabook.org/`.
All royalties from its sale will be donated to Amnesty International.

Front cover photo ©Kellar Wilson
Copyediting, cover design, and publishing support by Amy Brown: `http://amyrbrown.ca`

Revision Date: September 9, 2016

ISBN: 978-1-329-87127-4

Contents

Introduction ix
by Michael DiBernardo

1 Blockcode: A visual programming toolkit 1
by Dethe Elza

2 A Continuous Integration System 17
by Malini Das

3 Clustering by Consensus 33
by Dustin J. Mitchell

4 Contingent: A Fully Dynamic Build System 61
by Brandon Rhodes and Daniel Rocco

5 A Web Crawler With asyncio Coroutines 83
by A. Jesse Jiryu Davis and Guido van Rossum

6 Dagoba: an in-memory graph database 109
by Dann Toliver

7 DBDB: Dog Bed Database 141
by Taavi Burns

8 An Event-Driven Web Framework 153
by Leo Zovic

9 A Flow Shop Scheduler 175
by Dr. Christian Muise

10 An Archaeology-Inspired Database 191
by Yoav Rubin

11 Making Your Own Image Filters 215
by Cate Huston

12 A Python Interpreter Written in Python 243
by Allison Kaptur

13 A 3D Modeller 267
by Erick Dransch

14 A Simple Object Model 291
by Carl Friedrich Bolz

15 Optical Character Recognition (OCR) 309
by Marina Samuel

16 A Pedometer in the Real World 323
by Dessy Daskalov

17 The Same-Origin Policy 347
by Eunsuk Kang, Santiago Perez De Rosso, and Daniel Jackson

18 A Rejection Sampler 375
by Jessica B. Hamrick

19 Web Spreadsheet 395
by Audrey Tang

20 Static Analysis 409
by Leah Hanson

21 A Template Engine 431
by Ned Batchelder

22 A Simple Web Server 451
by Greg Wilson

Introduction

Michael DiBernardo

This is the fourth volume in the *Architecture of Open Source Applications* series, and the first to not feature the words "open source applications" anywhere in the title.

The first three volumes in the series were about big problems that big programs have to solve. For an engineer who is early in their career, it may be a challenge to understand and build upon programs that are much bigger than a few thousand lines of code, so, while big problems can be interesting to read about, they can also be challenging to learn from.

500 Lines or Less focuses on the design decisions that programmers make in the small when they are building something new. The programs you will read about in this book were all written from scratch for this purpose (although several of them were inspired by larger projects that the authors had worked on previously).

Before reading each chapter, we encourage you to first think about how you might solve the problem. What design considerations or constraints do you think the author is going to consider important? What abstractions do you expect to see? How do you think the problem is going to be decomposed? Then, when reading the chapter, try to identify what surprised you. It is our hope that you will learn more by doing this than by simply reading through each chapter from beginning to end.

Writing a useful program in fewer than 500 lines of source code—without resorting to cheap tricks—is a challenging exercise in itself; writing one to be read for pedagogical purposes when neatly rendered in a printed book is even tougher. As such, the editors have occasionally taken liberties with some of the source formatting when porting it into the book. The original source for each chapter can be found in the `code` subdirectory of its project folder.

We hope that the experiences of the authors in this book will help you grow out of your comfort zone in your own programming practice.

— Michael DiBernardo

Contributors

Michael DiBernardo (editorial): Michael DiBernardo is an engineer and director of delivery at Wave, and a past PyCon Canada chair. He writes at `mikedebo.ca`.

Amy Brown (editorial): Amy Brown is a freelance editor based in Toronto. She specializes in science and academic editing, and working with self-publishing authors. She co-edited the *Architecture of Open Source Applications* books with Greg Wilson.

Dethe Elza (Blockcode): Dethe is a geek dad, aesthetic programmer, mentor, and creator of the Waterbear visual programming tool. He co-hosts the Vancouver Maker Education Salons and wants to fill the world with robotic origami rabbits.

Malini Das (CI): Malini is a software engineer who is passionate about developing quickly (but safely!), and solving cross-functional problems. She has worked at Mozilla as a tools engineer and is currently honing her skills at Twitch.

Dustin J. Mitchell (Cluster): Dustin is an open source software developer and release engineer at Mozilla. He has worked on projects as varied as a host configuration system in Puppet, a Flask-based web framework, unit tests for firewall configurations, and a continuous integration framework in Twisted Python.

Daniel Rocco (Contingent): Daniel loves Python, coffee, craft, stout, object and system design, bourbon, teaching, trees, and Latin guitar. Thrilled that he gets to write Python for a living, he is always on the lookout for opportunities to learn from others in the community, and to contribute by sharing knowledge. He is a frequent speaker at PyAtl on introductory topics, testing, design, and shiny things; he loves seeing the spark of delight in people's eyes when someone shares a surprising or beautiful idea. Daniel lives in Atlanta with a microbiologist and four aspiring rocketeers.

Brandon Rhodes (Contingent): Brandon Rhodes started using Python in the late 1990s, and for 17 years has maintained the PyEphem library for amateur astronomers. He works at Dropbox, has taught Python programming courses for corporate clients, consulted on projects like the New England Wildflower Society's "Go Botany" Django site, and will be the chair of the PyCon conference in 2016 and 2017. Brandon believes that well-written code is a form of literature, that beautifully formatted code is a work of graphic design, and that correct code is one of the most transparent forms of thought.

A. Jesse Jiryu Davis (Crawler): Jesse is a staff engineer at MongoDB in New York. He wrote Motor, the async MongoDB Python driver, and he is the lead developer of the MongoDB C Driver and a member of the PyMongo team. He contributes to asyncio and Tornado. He writes at `emptysqua.re`.

Guido van Rossum (Crawler): Guido is the creator of Python, one of the major programming languages on and off the web. The Python community refers to him as the BDFL (Benevolent Dictator For Life), a title straight from a Monty Python skit.

Dann Toliver (Dagoba): Dann enjoys building things, like programming languages, databases, distributed systems, communities of smart friendly humans, and pony castles with his two-year-old.

Taavi Burns (DBDB): As the newest bass (and sometimes tenor) in Countermeasure, Taavi strives to break the mould…sometimes just by ignoring its existence. This is certainly true through the diversity of workplaces in his career: IBM (doing C and Perl), FreshBooks (all the things), Points.com (doing Python), and now at PagerDuty (doing Scala). Aside from that—when not gliding along on his Brompton folding bike—you might find him playing Minecraft with his son or engaging in parkour (or rock climbing, or other adventures) with his wife. He knits continental.

Leo Zovic: Leo (better known online as inaimathi) is a recovering graphic designer who has professionally written Scheme, Common Lisp, Erlang, Javascript, Haskell, Clojure, Go, Python,

PHP and C. He currently blogs about programming, plays board games and works at a Ruby-based startup in Toronto, Ontario.

Dr. Christian Muise (Flow shop): Dr. Muise is a Research Fellow with the Model-based Embedded and Robotic Systems group at MIT's Computer Science and Artificial Intelligence Laboratory. He is interested in a variety of topics including AI, data-driven projects, mapping, graph theory, and data visualization, as well as Celtic music, carving, soccer, and coffee.

Yoav Rubin (CircleDB): Yoav is a Senior Software Engineer at Microsoft, and prior to that was a Research Staff Member and a Master Inventor at IBM Research. He works now in the domain of data security in the cloud, and in the past his work focused on developing cloud- or web-based development environments. Yoav holds an MSc in Medical Research in the field of Neuroscience and BSc in Information Systems Engineering.

Cate Huston (Image filters): Cate is a developer and entrepreneur focused on mobile. She's lived and worked in the UK, Australia, Canada, China and the United States, as an engineer at Google, an Extreme Blue intern at IBM, and a ski instructor. Cate speaks internationally on mobile development, and her writing has been published on sites as varied as Lifehacker, The Daily Beast, The Eloquent Woman and Model View Culture. She co-curates Technically Speaking, blogs at Accidentally in Code and is `@catehstn` on Twitter.

Allison Kaptur (Interpreter): Allison is an engineer at Dropbox, where she helps maintain one of the largest networks of Python clients in the world. Before Dropbox, she was a facilitator at the Recurse Center, a writers' retreat for programmers in New York. She's spoken at PyCon North America about Python internals, and loves weird bugs.

Erick Dransch (Modeller): Erick is a software developer and 2D and 3D computer graphics enthusiast. He has worked on video games, 3D special effects software, and computer-aided design tools. If it involves simulating reality, chances are he'd like to learn more about it. You can find him online at `erickdransch.com`.

Carl Friedrich Bolz (Object model): Carl is a researcher at King's College London and is broadly interested in the implementation and optimization of all kinds of dynamic languages. He is one of the core authors of PyPy/RPython and has worked on implementations of Prolog, Racket, Smalltalk, PHP and Ruby.

Marina Samuel (OCR): Marina is an engineer at Mozilla and a current MSc student in Applied Computing (Artifical Intelligence) at the University of Toronto. She hopes to one day build robots that will take over the planet.

Dessy Daskalov (Pedometer): Dessy is an engineer by trade, an entrepreneur by passion, and a developer at heart. She's currently the CTO and co-founder of Nudge Rewards. When she's not busy building product with her team, she can be found teaching others to code, attending or hosting a Toronto tech event, and online at `dessydaskalov.com` and `@dess_e`.

Eunsuk Kang (Same-origin policy): Eunsuk is a PhD candidate and a member of the Software Design Group at MIT. He received his SM (Master of Science) in Computer Science from MIT (2010), and a Bachelor of Software Engineering from the University of Waterloo (2007). His research projects have focused on developing tools and techniques for software modeling and verification, with applications to security and safety-critical systems.

Santiago Perez (Same-origin policy): Santiago is a PhD student in the Software Design Group at MIT. He received his SM in Computer Science from MIT (2015), and an undergraduate degree from ITBA (2011). He used to work at Google, developing frameworks and tools to make engineers more productive (2012). He currently spends most of his time thinking about design and version control.

Daniel Jackson (Same-origin policy): Daniel is a professor in the Department of Electrical Engineering and Computer Science at MIT, and leads the Software Design Group in the Computer

Science and Artificial Intelligence Laboratory. He received an MA from Oxford University (1984) in Physics, and his SM (1988) and PhD (1992) in Computer Science from MIT. He was a software engineer for Logica UK Ltd. (1984-1986), Assistant Professor of Computer Science at Carnegie Mellon University (1992-1997), and has been at MIT since 1997. He has broad interests in software engineering, especially in development methods, design and specification, formal methods, and safety-critical systems.

Jessica B. Hamrick (Sampler): Jess is a PhD student at UC Berkeley where she studies human cognition by combining probabilistic models from machine learning with behavioral experiments from cognitive science. In her spare time, Jess is a core contributor to IPython and Jupyter. She also holds a BS and MEng in Computer Science from MIT.

Audrey Tang (Spreadsheet): A self-educated programmer and translator, Audrey works with Apple as an independent contractor on cloud service localization and natural language technologies. Audrey has previously designed and led the first working Perl 6 implementation, and served in computer language design committees for Haskell, Perl 5, and Perl 6. Currently Audrey is a full-time g0v contributor and leads Taiwan's first e-Rulemaking project.

Leah Hanson (Static analysis): Leah Hanson is a proud alum of Hacker School and loves helping people learn about Julia. She blogs at `blog.leahhanson.us` and tweets at `@astrieanna`.

Ned Batchelder (Template engine): Ned is a software engineer with a long career, currently working at edX to build open source software to educate the world. He's the maintainer of coverage.py, an organizer of Boston Python, and has spoken at many PyCons. He blogs at `nedbatchelder.com`. He once had dinner at the White House.

Greg Wilson (Web server): Greg is the founder of Software Carpentry, a crash course in computing skills for scientists and engineers. He has worked for 30 years in both industry and academia, and is the author or editor of several books on computing, including the 2008 Jolt Award winner *Beautiful Code* and the first two volumes of *The Architecture of Open Source Applications*. Greg received a PhD in Computer Science from the University of Edinburgh in 1993.

Acknowledgments

The *Architecture of Open Source Applications* series would not exist without the hard work of Amy Brown and Greg Wilson. This particular book would not have been possible without the incredible efforts of our army of technical reviewers:

Amber Yust
Andrew Gwozdziewycz
Andrew Kuchling
Andrew Svetlov
Andy Shen
Anton Beloglazov
Ben Trofatter
Borys Pierov
Carise Fernandez
Charles Stanhope
Chris AtLee
Chris Seaton
Cyryl Płotnicki-Chudyk
Dan Langer
Dan Shapiro
David Pokorny
Eric Bouwers
Frederic De Groef
Graham Lee
Gregory Eric Sanderson
James O'Beirne
Jan de Baat
Jana Beck
Jessica McKellar
Jo Van Eyck
Joel Crocker
Johan Thelin
Johannes Fürmann
John Morrissey
Joseph Kaptur
Josh Crompton
Joshua T. Corbin
Kevin Huang
Maggie Zhou
Marc Towler
Marcin Milewski
Marco Lancini
Mark Reid
Matthias Bussonnier
Max Mautner
Meggin Kearney
Mike Aquino
Natalie Black
Nick Presta
Nikhil Almeida
Nolan Prescott
Paul Martin
Piotr Banaszkiewicz
Preston Holmes
Pulkit Sethi
Rail Aliiev
Ronen Narkis
Rose Ames
Sina Jahan
Stefan Turalski
William Lachance

Chris Seaton, John Morrissey, and Natalie Black deserve extended thanks for going above and beyond in their technical reviewing. The quantity and depth of their reviews was instrumental in moving the book forward at several sticking points.

We are very grateful to PagerDuty for their financial support.

Contributing

If you'd like to report errors or translate the content into other languages, please open an issue at `github.com/aosabook/500lines/` or contact us at `aosa@aosabook.org`.

Blockcode: A visual programming toolkit

Dethe Elza

In block-based programming languages, you write programs by dragging and connecting blocks that represent parts of the program. Block-based languages differ from conventional programming languages, in which you type words and symbols.

Learning a programming language can be difficult because they are extremely sensitive to even the slightest of typos. Most programming languages are case-sensitive, have obscure syntax, and will refuse to run if you get so much as a semicolon in the wrong place—or worse, leave one out. Further, most programming languages in use today are based on English and their syntax cannot be localized.

In contrast, a well-done block language can eliminate syntax errors completely. You can still create a program which does the wrong thing, but you cannot create one with the wrong syntax: the blocks just won't fit that way. Block languages are more discoverable: you can see all the constructs and libraries of the language right in the list of blocks. Further, blocks can be localized into any human language without changing the meaning of the programming language.

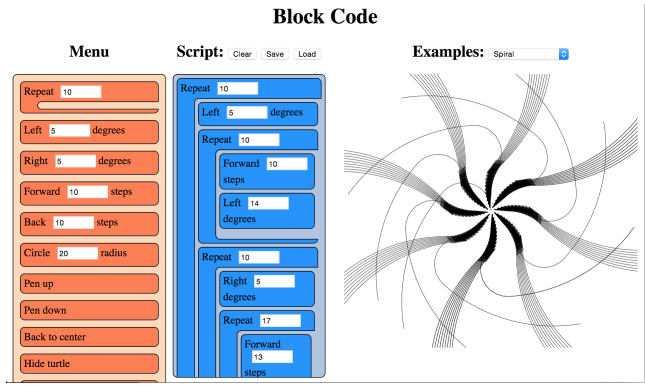

Figure 1.1: The Blockcode IDE in use

Block-based languages have a long history, with some of the prominent ones being Lego Mindstorms[1], Alice3D[2], StarLogo[3], and especially Scratch[4]. There are several tools for block-based

[1] http://www.lego.com/en-us/mindstorms/
[2] http://www.alice.org/index.php
[3] http://education.mit.edu/projects/starlogo-tng
[4] http://scratch.mit.edu/

programming on the web as well: Blockly[5], AppInventor[6], Tynker[7], and many more[8].

The code in this chapter is loosely based on the open-source project Waterbear[9], which is not a language but a tool for wrapping existing languages with a block-based syntax. Advantages of such a wrapper include the ones noted above: eliminating syntax errors, visual display of available components, ease of localization. Additionally, visual code can sometimes be easier to read and debug, and blocks can be used by pre-typing children. (We could even go further and put icons on the blocks, either in conjunction with the text names or instead of them, to allow pre-literate children to write programs, but we don't go that far in this example.)

The choice of turtle graphics for this language goes back to the Logo language, which was created specifically to teach programming to children. Several of the block-based languages above include turtle graphics, and it is a small enough domain to be able to capture in a tightly constrained project such as this.

If you would like to get a feel for what a block-based-language is like, you can experiment with the program that is built in this chapter from author's GitHub repository[10].

1.1 Goals and Structure

I want to accomplish a couple of things with this code. First and foremost, I want to implement a block language for turtle graphics, with which you can write code to create images through simple dragging-and-dropping of blocks, using as simple a structure of HTML, CSS, and JavaScript as possible. Second, but still important, I want to show how the blocks themselves can serve as a framework for other languages besides our mini turtle language.

To do this, we encapsulate everything that is specific to the turtle language into one file (`turtle.js`) that we can easily swap with another file. Nothing else should be specific to the turtle language; the rest should just be about handling the blocks (`blocks.js` and `menu.js`) or be generally useful web utilities (`util.js`, `drag.js`, `file.js`). That is the goal, although to maintain the small size of the project, some of those utilities are less general-purpose and more specific to their use with the blocks.

One thing that struck me when writing a block language was that the language is its own IDE. You can't just code up blocks in your favourite text editor; the IDE has to be designed and developed in parallel with the block language. This has some pros and cons. On the plus side, everyone will use a consistent environment and there is no room for religious wars about what editor to use. On the downside, it can be a huge distraction from building the block language itself.

The Nature of Scripts

A Blockcode script, like a script in any language (whether block- or text-based), is a sequence of operations to be followed. In the case of Blockcode the script consists of HTML elements which are iterated over, and which are each associated with a particular JavaScript function which will be run when that block's turn comes. Some blocks can contain (and are responsible for running) other blocks, and some blocks can contain numeric arguments which are passed to the functions.

[5] `https://developers.google.com/blockly/`
[6] `http://appinventor.mit.edu/explore/`
[7] `http://www.tynker.com/`
[8] `http://en.wikipedia.org/wiki/Visual_programming_language`
[9] `http://waterbearlang.com/`
[10] `https://dethe.github.io/500lines/blockcode/`

In most (text-based) languages, a script goes through several stages: a lexer converts the text into recognized tokens, a parser organizes the tokens into an abstract syntax tree, then depending on the language the program may be compiled into machine code or fed into an interpreter. That's a simplification; there can be more steps. For Blockcode, the layout of the blocks in the script area already represents our abstract syntax tree, so we don't have to go through the lexing and parsing stages. We use the Visitor pattern to iterate over those blocks and call predefined JavaScript functions associated with each block to run the program.

There is nothing stopping us from adding additional stages to be more like a traditional language. Instead of simply calling associated JavaScript functions, we could replace `turtle.js` with a block language that emits byte codes for a different virtual machine, or even C++ code for a compiler. Block languages exist (as part of the Waterbear project) for generating Java robotics code, for programming Arduino, and for scripting Minecraft running on Raspberry Pi.

Web Applications

In order to make the tool available to the widest possible audience, it is web-native. It's written in HTML, CSS, and JavaScript, so it should work in most browsers and platforms.

Modern web browsers are powerful platforms, with a rich set of tools for building great apps. If something about the implementation became too complex, I took that as a sign that I wasn't doing it "the web way" and, where possible, tried to re-think how to better use the browser tools.

An important difference between web applications and traditional desktop or server applications is the lack of a `main()` or other entry point. There is no explicit run loop because that is already built into the browser and implicit on every web page. All our code will be parsed and executed on load, at which point we can register for events we are interested in for interacting with the user. After the first run, all further interaction with our code will be through callbacks we set up and register, whether we register those for events (like mouse movement), timeouts (fired with the periodicity we specify), or frame handlers (called for each screen redraw, generally 60 frames per second). The browser does not expose full-featured threads either (only shared-nothing web workers).

1.2 Stepping Through the Code

I've tried to follow some conventions and best practices throughout this project. Each JavaScript file is wrapped in a function to avoid leaking variables into the global environment. If it needs to expose variables to other files it will define a single global per file, based on the filename, with the exposed functions in it. This will be near the end of the file, followed by any event handlers set by that file, so you can always glance at the end of a file to see what events it handles and what functions it exposes.

The code style is procedural, not object-oriented or functional. We could do the same things in any of these paradigms, but that would require more setup code and wrappers to impose on what exists already for the DOM. Recent work on Custom Elements[11] make it easier to work with the DOM in an OO way, and there has been a lot of great writing on Functional JavaScript[12], but either would require a bit of shoe-horning, so it felt simpler to keep it procedural.

There are eight source files in this project, but `index.html` and `blocks.css` are basic structure and style for the app and won't be discussed. Two of the JavaScript files won't be discussed in any detail either: `util.js` contains some helpers and serves as a bridge between different browser

[11] `http://webcomponents.org/`
[12] `https://leanpub.com/javascript-allonge/read`

implementations—similar to a library like jQuery but in less than 50 lines of code. `file.js` is a similar utility used for loading and saving files and serializing scripts.

These are the remaining files:

- `block.js` is the abstract representation of a block-based language.
- `drag.js` implements the key interaction of the language: allowing the user to drag blocks from a list of available blocks (the "menu") to assemble them into a program (the "script").
- `menu.js` has some helper code and is also responsible for actually running the user's program.
- `turtle.js` defines the specifics of our block language (turtle graphics) and initializes its specific blocks. This is the file that would be replaced in order to create a different block language.

`blocks.js`blocks.js

Each block consists of a few HTML elements, styled with CSS, with some JavaScript event handlers for dragging-and-dropping and modifying the input argument. The `blocks.js` file helps to create and manage these groupings of elements as single objects. When a type of block is added to the block menu, it is associated with a JavaScript function to implement the language, so each block in the script has to be able to find its associated function and call it when the script runs.

Figure 1.2: An example block

Blocks have two optional bits of structure. They can have a single numeric parameter (with a default value), and they can be a container for other blocks. These are hard limits to work with, but would be relaxed in a larger system. In Waterbear there are also expression blocks which can be passed in as parameters; multiple parameters of a variety of types are supported. Here in the land of tight constraints we'll see what we can do with just one type of parameter.

```
<!-- The HTML structure of a block -->
<div class="block" draggable="true" data-name="Right">
    Right
    <input type="number" value="5">
    degrees
</div>
```

It's important to note that there is no real distinction between blocks in the menu and blocks in the script. Dragging treats them slightly differently based on where they are being dragged from, and when we run a script it only looks at the blocks in the script area, but they are fundamentally the same structures, which means we can clone the blocks when dragging from the menu into the script.

The `createBlock(name, value, contents)` function returns a block as a DOM element populated with all internal elements, ready to insert into the document. This can be used to create blocks for the menu, or for restoring script blocks saved in files or `localStorage`. While it is flexible this way, it is built specifically for the Blockcode "language" and makes assumptions about it, so if there is a value it assumes the value represents a numeric argument and creates an input of type "number". Since this is a limitation of the Blockcode, this is fine, but if we were to extend the blocks to support other types of arguments, or more than one argument, the code would have to change.

```
function createBlock(name, value, contents){
    var item = elem('div',
        {'class': 'block', draggable: true, 'data-name': name},
        [name]
    );
    if (value !== undefined && value !== null){
        item.appendChild(elem('input', {type: 'number', value: value}));
    }
    if (Array.isArray(contents)){
        item.appendChild(
            elem('div', {'class': 'container'}, contents.map(function(block){
            return createBlock.apply(null, block);
        })));
    }else if (typeof contents === 'string'){
        // Add units (degrees, etc.) specifier
        item.appendChild(document.createTextNode(' ' + contents));
    }
    return item;
}
```

We have some utilities for handling blocks as DOM elements:

- `blockContents(block)` retrieves the child blocks of a container block. It always returns a list if called on a container block, and always returns null on a simple block
- `blockValue(block)` returns the numerical value of the input on a block if the block has an input field of type number, or null if there is no input element for the block
- `blockScript(block)` will return a structure suitable for serializing with JSON, to save blocks in a form they can easily be restored from
- `runBlocks(blocks)` is a handler that runs each block in an array of blocks

```
function blockContents(block){
    var container = block.querySelector('.container');
    return container ? [].slice.call(container.children) : null;
}

function blockValue(block){
    var input = block.querySelector('input');
    return input ? Number(input.value) : null;
}

function blockUnits(block){
    if (block.children.length > 1 &&
        block.lastChild.nodeType === Node.TEXT_NODE &&
        block.lastChild.textContent){
        return block.lastChild.textContent.slice(1);
    }
}

function blockScript(block){
    var script = [block.dataset.name];
    var value = blockValue(block);
    if (value !== null){
```

```
        script.push(blockValue(block));
    }
    var contents = blockContents(block);
    var units = blockUnits(block);
    if (contents){script.push(contents.map(blockScript));}
    if (units){script.push(units);}
    return script.filter(function(notNull){ return notNull !== null; });
}

function runBlocks(blocks){
    blocks.forEach(function(block){ trigger('run', block); });
}
```

drag.jsdrag.js

The purpose of drag.js is to turn static blocks of HTML into a dynamic programming language by implementing interactions between the menu section of the view and the script section. The user builds their program by dragging blocks from the menu into the script, and the system runs the blocks in the script area.

We're using HTML5 drag-and-drop; the specific JavaScript event handlers it requires are defined here. (For more information on using HTML5 drag-and-drop, see Eric Bidleman's article[13].) While it is nice to have built-in support for drag-and-drop, it does have some oddities and some pretty major limitations, like not being implemented in any mobile browser at the time of this writing.

We define some variables at the top of the file. When we're dragging, we'll need to reference these from different stages of the dragging callback dance.

```
var dragTarget = null; // Block we're dragging
var dragType = null; // Are we dragging from the menu or from the script?
var scriptBlocks = []; // Blocks in the script, sorted by position
```

Depending on where the drag starts and ends, drop will have different effects:

- If dragging from script to menu, delete dragTarget (remove block from script).
- If dragging from script to script, move dragTarget (move an existing script block).
- If dragging from menu to script, copy dragTarget (insert new block in script).
- If dragging from menu to menu, do nothing.

During the dragStart(evt) handler we start tracking whether the block is being copied from the menu or moved from (or within) the script. We also grab a list of all the blocks in the script which are not being dragged, to use later. The evt.dataTransfer.setData call is used for dragging between the browser and other applications (or the desktop), which we're not using, but have to call anyway to work around a bug.

```
function dragStart(evt){
    if (!matches(evt.target, '.block')) return;
    if (matches(evt.target, '.menu .block')){
        dragType = 'menu';
    }else{
        dragType = 'script';
```

[13] http://www.html5rocks.com/en/tutorials/dnd/basics/

```
    }
    evt.target.classList.add('dragging');
    dragTarget = evt.target;
    scriptBlocks = [].slice.call(
        document.querySelectorAll('.script .block:not(.dragging)'));
    // For dragging to take place in Firefox, we have to set this, even if
    // we don't use it
    evt.dataTransfer.setData('text/html', evt.target.outerHTML);
    if (matches(evt.target, '.menu .block')){
        evt.dataTransfer.effectAllowed = 'copy';
    }else{
        evt.dataTransfer.effectAllowed = 'move';
    }
}
```

While we are dragging, the `dragenter`, `dragover`, and `dragout` events give us opportunities to add visual cues by highlighting valid drop targets, etc. Of these, we only make use of `dragover`.

```
function dragOver(evt){
    if (!matches(evt.target, '.menu, .menu *, .script, .script *, .content')) {
        return;
    }
    // Necessary. Allows us to drop.
    if (evt.preventDefault) { evt.preventDefault(); }
    if (dragType === 'menu'){
        // See the section on the DataTransfer object.
        evt.dataTransfer.dropEffect = 'copy';
    }else{
        evt.dataTransfer.dropEffect = 'move';
    }
    return false;
}
```

When we release the mouse, we get a `drop` event. This is where the magic happens. We have to check where we dragged from (set back in `dragStart`) and where we have dragged to. Then we either copy the block, move the block, or delete the block as needed. We fire off some custom events using `trigger()` (defined in `util.js`) for our own use in the block logic, so we can refresh the script when it changes.

```
function drop(evt){
    if (!matches(evt.target, '.menu, .menu *, .script, .script *')) return;
    var dropTarget = closest(
        evt.target, '.script .container, .script .block, .menu, .script');
    var dropType = 'script';
    if (matches(dropTarget, '.menu')){ dropType = 'menu'; }
    // stops the browser from redirecting.
    if (evt.stopPropagation) { evt.stopPropagation(); }
    if (dragType === 'script' && dropType === 'menu'){
        trigger('blockRemoved', dragTarget.parentElement, dragTarget);
        dragTarget.parentElement.removeChild(dragTarget);
    }else if (dragType ==='script' && dropType === 'script'){
```

```
            if (matches(dropTarget, '.block')){
                dropTarget.parentElement.insertBefore(
                    dragTarget, dropTarget.nextSibling);
            }else{
                dropTarget.insertBefore(dragTarget, dropTarget.firstChildElement);
            }
            trigger('blockMoved', dropTarget, dragTarget);
        }else if (dragType === 'menu' && dropType === 'script'){
            var newNode = dragTarget.cloneNode(true);
            newNode.classList.remove('dragging');
            if (matches(dropTarget, '.block')){
                dropTarget.parentElement.insertBefore(
                    newNode, dropTarget.nextSibling);
            }else{
                dropTarget.insertBefore(newNode, dropTarget.firstChildElement);
            }
            trigger('blockAdded', dropTarget, newNode);
        }
    }
```

The `dragEnd(evt)` is called when we mouse up, but after we handle the `drop` event. This is where we can clean up, remove classes from elements, and reset things for the next drag.

```
    function _findAndRemoveClass(klass){
        var elem = document.querySelector('.' + klass);
        if (elem){ elem.classList.remove(klass); }
    }

    function dragEnd(evt){
        _findAndRemoveClass('dragging');
        _findAndRemoveClass('over');
        _findAndRemoveClass('next');
    }
```

`menu.js`menu.js

The file `menu.js` is where blocks are associated with the functions that are called when they run, and contains the code for actually running the script as the user builds it up. Every time the script is modified, it is re-run automatically.

"Menu" in this context is not a drop-down (or pop-up) menu, like in most applications, but is the list of blocks you can choose for your script. This file sets that up, and starts the menu off with a looping block that is generally useful (and thus not part of the turtle language itself). This is kind of an odds-and-ends file, for things that may not fit anywhere else.

Having a single file to gather random functions in is useful, especially when an architecture is under development. My theory of keeping a clean house is to have designated places for clutter, and that applies to building a program architecture too. One file or module becomes the catch-all for things that don't have a clear place to fit in yet. As this file grows it is important to watch for emerging patterns: several related functions can be spun off into a separate module (or joined together into a more general function). You don't want the catch-all to grow indefinitely, but only to be a temporary holding place until you figure out the right way to organize the code.

We keep around references to `menu` and `script` because we use them a lot; no point hunting through the DOM for them over and over. We'll also use `scriptRegistry`, where we store the scripts of blocks in the menu. We use a very simple name-to-script mapping which does not support either multiple menu blocks with the same name or renaming blocks. A more complex scripting environment would need something more robust.

We use `scriptDirty` to keep track of whether the script has been modified since the last time it was run, so we don't keep trying to run it constantly.

```
var menu = document.querySelector('.menu');
var script = document.querySelector('.script');
var scriptRegistry = {};
var scriptDirty = false;
```

When we want to notify the system to run the script during the next frame handler, we call `runSoon()` which sets the `scriptDirty` flag to `true`. The system calls `run()` on every frame, but returns immediately unless `scriptDirty` is set. When `scriptDirty` is set, it runs all the script blocks, and also triggers events to let the specific language handle any tasks it needs before and after the script is run. This decouples the blocks-as-toolkit from the turtle language to make the blocks re-usable (or the language pluggable, depending how you look at it).

As part of running the script, we iterate over each block, calling `runEach(evt)` on it, which sets a class on the block, then finds and executes its associated function. If we slow things down, you should be able to watch the code execute as each block highlights to show when it is running.

The `requestAnimationFrame` method below is provided by the browser for animation. It takes a function which will be called for the next frame to be rendered by the browser (at 60 frames per second) after the call is made. How many frames we actually get depends on how fast we can get work done in that call.

```
function runSoon(){ scriptDirty = true; }

function run(){
    if (scriptDirty){
        scriptDirty = false;
        Block.trigger('beforeRun', script);
        var blocks = [].slice.call(
            document.querySelectorAll('.script > .block'));
        Block.run(blocks);
        Block.trigger('afterRun', script);
    }else{
        Block.trigger('everyFrame', script);
    }
    requestAnimationFrame(run);
}
requestAnimationFrame(run);

function runEach(evt){
    var elem = evt.target;
    if (!matches(elem, '.script .block')) return;
    if (elem.dataset.name === 'Define block') return;
    elem.classList.add('running');
    scriptRegistry[elem.dataset.name](elem);
```

```
    elem.classList.remove('running');
}
```

We add blocks to the menu using `menuItem(name, fn, value, contents)` which takes a normal block, associates it with a function, and puts in the menu column.

```
function menuItem(name, fn, value, units){
    var item = Block.create(name, value, units);
    scriptRegistry[name] = fn;
    menu.appendChild(item);
    return item;
}
```

We define `repeat(block)` here, outside of the turtle language, because it is generally useful in different languages. If we had blocks for conditionals and reading and writing variables they could also go here, or into a separate trans-language module, but right now we only have one of these general-purpose blocks defined.

```
function repeat(block){
    var count = Block.value(block);
    var children = Block.contents(block);
    for (var i = 0; i < count; i++){
        Block.run(children);
    }
}
menuItem('Repeat', repeat, 10, []);
```

`turtle.js`turtle.js

`turtle.js` is the implementation of the turtle block language. It exposes no functions to the rest of the code, so nothing else can depend on it. This way we can swap out the one file to create a new block language and know nothing in the core will break.

Turtle programming is a style of graphics programming, first popularized by Logo, where you have an imaginary turtle carrying a pen walking on the screen. You can tell the turtle to pick up the pen (stop drawing, but still move), put the pen down (leaving a line everywhere it goes), move forward a number of steps, or turn a number of degrees. Just those commands, combined with looping, can create amazingly intricate images.

In this version of turtle graphics we have a few extra blocks. Technically we don't need both `turn right` and `turn left` because you can have one and get the other with negative numbers. Likewise `move back` can be done with `move forward` and negative numbers. In this case it felt more balanced to have both.

The image above was formed by putting two loops inside another loop and adding a `move forward` and `turn right` to each loop, then playing with the parameters interactively until I liked the image that resulted.

```
var PIXEL_RATIO = window.devicePixelRatio || 1;
var canvasPlaceholder = document.querySelector('.canvas-placeholder');
var canvas = document.querySelector('.canvas');
var script = document.querySelector('.script');
var ctx = canvas.getContext('2d');
```

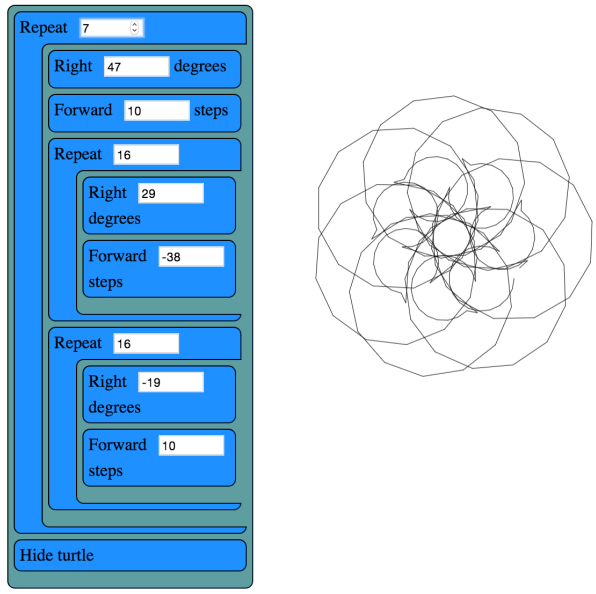

Figure 1.3: Example of Turtle code running

```
var cos = Math.cos, sin = Math.sin, sqrt = Math.sqrt, PI = Math.PI;
var DEGREE = PI / 180;
var WIDTH, HEIGHT, position, direction, visible, pen, color;
```

The `reset()` function clears all the state variables to their defaults. If we were to support multiple turtles, these variables would be encapsulated in an object. We also have a utility, `deg2rad(deg)`, because we work in degrees in the UI, but we draw in radians. Finally, `drawTurtle()` draws the turtle itself. The default turtle is simply a triangle, but you could override this to draw a more aesthetically-pleasing turtle.

Note that `drawTurtle` uses the same primitive operations that we define to implement the turtle drawing. Sometimes you don't want to reuse code at different abstraction layers, but when the meaning is clear it can be a big win for code size and performance.

```
function reset(){
    recenter();
    direction = deg2rad(90); // facing "up"
    visible = true;
    pen = true; // when pen is true we draw, otherwise we move without drawing
    color = 'black';
}

function deg2rad(degrees){ return DEGREE * degrees; }

function drawTurtle(){
    var userPen = pen; // save pen state
    if (visible){
        penUp(); _moveForward(5); penDown();
        _turn(-150); _moveForward(12);
```

```
        _turn(-120); _moveForward(12);
        _turn(-120); _moveForward(12);
        _turn(30);
        penUp(); _moveForward(-5);
        if (userPen){
            penDown(); // restore pen state
        }
    }
}
```

We have a special block to draw a circle with a given radius at the current mouse position. We special-case `drawCircle` because, while you can certainly draw a circle by repeating `MOVE 1 RIGHT 1` 360 times, controlling the size of the circle is very difficult that way.

```
function drawCircle(radius){
    // Math for this is from http://www.mathopenref.com/polygonradius.html
    var userPen = pen; // save pen state
    if (visible){
        penUp(); _moveForward(-radius); penDown();
        _turn(-90);
        var steps = Math.min(Math.max(6, Math.floor(radius / 2)), 360);
        var theta = 360 / steps;
        var side = radius * 2 * Math.sin(Math.PI / steps);
        _moveForward(side / 2);
        for (var i = 1; i < steps; i++){
            _turn(theta); _moveForward(side);
        }
        _turn(theta); _moveForward(side / 2);
        _turn(90);
        penUp(); _moveForward(radius); penDown();
        if (userPen){
            penDown(); // restore pen state
        }
    }
}
```

Our main primitive is `moveForward`, which has to handle some elementary trigonometry and check whether the pen is up or down.

```
function _moveForward(distance){
    var start = position;
    position = {
        x: cos(direction) * distance * PIXEL_RATIO + start.x,
        y: -sin(direction) * distance * PIXEL_RATIO + start.y
    };
    if (pen){
        ctx.lineStyle = color;
        ctx.beginPath();
        ctx.moveTo(start.x, start.y);
        ctx.lineTo(position.x, position.y);
        ctx.stroke();
    }
```

```
}
```

Most of the rest of the turtle commands can be easily defined in terms of what we've built above.

```
function penUp(){ pen = false; }
function penDown(){ pen = true; }
function hideTurtle(){ visible = false; }
function showTurtle(){ visible = true; }
function forward(block){ _moveForward(Block.value(block)); }
function back(block){ _moveForward(-Block.value(block)); }
function circle(block){ drawCircle(Block.value(block)); }
function _turn(degrees){ direction += deg2rad(degrees); }
function left(block){ _turn(Block.value(block)); }
function right(block){ _turn(-Block.value(block)); }
function recenter(){ position = {x: WIDTH/2, y: HEIGHT/2}; }
```

When we want a fresh slate, the `clear` function restores everything back to where we started.

```
function clear(){
    ctx.save();
    ctx.fillStyle = 'white';
    ctx.fillRect(0,0,WIDTH,HEIGHT);
    ctx.restore();
    reset();
    ctx.moveTo(position.x, position.y);
}
```

When this script first loads and runs, we use our `reset` and `clear` to initialize everything and draw the turtle.

```
onResize();
clear();
drawTurtle();
```

Now we can use the functions above, with the `Menu.item` function from `menu.js`, to make blocks for the user to build scripts from. These are dragged into place to make the user's programs.

```
Menu.item('Left', left, 5, 'degrees');
Menu.item('Right', right, 5, 'degrees');
Menu.item('Forward', forward, 10, 'steps');
Menu.item('Back', back, 10, 'steps');
Menu.item('Circle', circle, 20, 'radius');
Menu.item('Pen up', penUp);
Menu.item('Pen down', penDown);
Menu.item('Back to center', recenter);
Menu.item('Hide turtle', hideTurtle);
Menu.item('Show turtle', showTurtle);
```

1.3 Lessons Learned

Why Not Use MVC?

Model-View-Controller (MVC) was a good design choice for Smalltalk programs in the ’80s and it can work in some variation or other for web apps, but it isn’t the right tool for every problem. All the state (the “model” in MVC) is captured by the block elements in a block language anyway, so replicating it into Javascript has little benefit unless there is some other need for the model (if we were editing shared, distributed code, for instance).

An early version of Waterbear went to great lengths to keep the model in JavaScript and sync it with the DOM, until I noticed that more than half the code and 90% of the bugs were due to keeping the model in sync with the DOM. Eliminating the duplication allowed the code to be simpler and more robust, and with all the state on the DOM elements, many bugs could be found simply by looking at the DOM in the developer tools. So in this case there is little benefit to building further separation of MVC than we already have in HTML/CSS/JavaScript.

Toy Changes Can Lead to Real Changes

Building a small, tightly scoped version of the larger system I work on has been an interesting exercise. Sometimes in a large system there are things you are hesitant to change because they affect too many other things. In a tiny, toy version you can experiment freely and learn things which you can then take back to the larger system. For me, the larger system is Waterbear and this project has had a huge impact on the way Waterbear is structured.

Small Experiments Make Failure OK

Some of the experiments I was able to do with this stripped-down block language were:

- using HTML5 drag-and-drop,
- running blocks directly by iterating through the DOM calling associated functions,
- separating the code that runs cleanly from the HTML DOM,
- simplified hit testing while dragging,
- building our own tiny vector and sprite libraries (for the game blocks), and
- “live coding” where the results are shown whenever you change the block script.

The thing about experiments is that they do not have to succeed. We tend to gloss over failures and dead ends in our work, where failures are punished instead of treated as important vehicles for learning, but failures are essential if you are going to push forward. While I did get the HTML5 drag-and-drop working, the fact that it isn’t supported at all on any mobile browser means it is a non-starter for Waterbear. Separating the code out and running code by iterating through the blocks worked so well that I’ve already begun bringing those ideas to Waterbear, with excellent improvements in testing and debugging. The simplified hit testing, with some modifications, is also coming back to Waterbear, as are the tiny vector and sprite libraries. Live coding hasn’t made it to Waterbear yet, but once the current round of changes stabilizes I may introduce it.

What Are We Trying to Build, Really?

Building a small version of a bigger system puts a sharp focus on what the important parts really are. Are there bits left in for historical reasons that serve no purpose (or worse, distract from the

purpose)? Are there features no-one uses but you have to pay to maintain? Could the user interface be streamlined? All these are great questions to ask while making a tiny version. Drastic changes, like re-organizing the layout, can be made without worrying about the ramifications cascading through a more complex system, and can even guide refactoring the complex system.

A Program is a Process, Not a Thing

There are things I wasn't able to experiment with in the scope of this project that I may use the blockcode codebase to test out in the future. It would be interesting to create "function" blocks which create new blocks out of existing blocks. Implementing undo/redo would be simpler in a constrained environment. Making blocks accept multiple arguments without radically expanding the complexity would be useful. And finding various ways to share block scripts online would bring the webbiness of the tool full circle.

A Web Crawler With asyncio Coroutines

A. Jesse Jiryu Davis and Guido van Rossum

5.1 Introduction

Classical computer science emphasizes efficient algorithms that complete computations as quickly as possible. But many networked programs spend their time not computing, but holding open many connections that are slow, or have infrequent events. These programs present a very different challenge: to wait for a huge number of network events efficiently. A contemporary approach to this problem is asynchronous I/O, or "async".

This chapter presents a simple web crawler. The crawler is an archetypal async application because it waits for many responses, but does little computation. The more pages it can fetch at once, the sooner it completes. If it devotes a thread to each in-flight request, then as the number of concurrent requests rises it will run out of memory or other thread-related resource before it runs out of sockets. It avoids the need for threads by using asynchronous I/O.

We present the example in three stages. First, we show an async event loop and sketch a crawler that uses the event loop with callbacks: it is very efficient, but extending it to more complex problems would lead to unmanageable spaghetti code. Second, therefore, we show that Python coroutines are both efficient and extensible. We implement simple coroutines in Python using generator functions. In the third stage, we use the full-featured coroutines from Python's standard "asyncio" library[1], and coordinate them using an async queue.

5.2 The Task

A web crawler finds and downloads all pages on a website, perhaps to archive or index them. Beginning with a root URL, it fetches each page, parses it for links to unseen pages, and adds these to a queue. It stops when it fetches a page with no unseen links and the queue is empty.

We can hasten this process by downloading many pages concurrently. As the crawler finds new links, it launches simultaneous fetch operations for the new pages on separate sockets. It parses responses as they arrive, adding new links to the queue. There may come some point of diminishing returns where too much concurrency degrades performance, so we cap the number of concurrent requests, and leave the remaining links in the queue until some in-flight requests complete.

[1] Guido introduced the standard asyncio library, called "Tulip" then, at PyCon 2013.

5.3 The Traditional Approach

How do we make the crawler concurrent? Traditionally we would create a thread pool. Each thread would be in charge of downloading one page at a time over a socket. For example, to download a page from `xkcd.com`:

```
def fetch(url):
    sock = socket.socket()
    sock.connect(('xkcd.com', 80))
    request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(url)
    sock.send(request.encode('ascii'))
    response = b''
    chunk = sock.recv(4096)
    while chunk:
        response += chunk
        chunk = sock.recv(4096)

    # Page is now downloaded.
    links = parse_links(response)
    q.add(links)
```

By default, socket operations are *blocking*: when the thread calls a method like `connect` or `recv`, it pauses until the operation completes.[2] Consequently to download many pages at once, we need many threads. A sophisticated application amortizes the cost of thread-creation by keeping idle threads in a thread pool, then checking them out to reuse them for subsequent tasks; it does the same with sockets in a connection pool.

And yet, threads are expensive, and operating systems enforce a variety of hard caps on the number of threads a process, user, or machine may have. On Jesse's system, a Python thread costs around 50k of memory, and starting tens of thousands of threads causes failures. If we scale up to tens of thousands of simultaneous operations on concurrent sockets, we run out of threads before we run out of sockets. Per-thread overhead or system limits on threads are the bottleneck.

In his influential article "The C10K problem"[3], Dan Kegel outlines the limitations of multithreading for I/O concurrency. He begins,

> It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the web is a big place now.

Kegel coined the term "C10K" in 1999. Ten thousand connections sounds dainty now, but the problem has changed only in size, not in kind. Back then, using a thread per connection for C10K was impractical. Now the cap is orders of magnitude higher. Indeed, our toy web crawler would work just fine with threads. Yet for very large scale applications, with hundreds of thousands of connections, the cap remains: there is a limit beyond which most systems can still create sockets, but have run out of threads. How can we overcome this?

[2] Even calls to `send` can block, if the recipient is slow to acknowledge outstanding messages and the system's buffer of outgoing data is full.

[3] `http://www.kegel.com/c10k.html`

5.4 Async

Asynchronous I/O frameworks do concurrent operations on a single thread using *non-blocking* sockets. In our async crawler, we set the socket non-blocking before we begin to connect to the server:

```
sock = socket.socket()
sock.setblocking(False)
try:
    sock.connect(('xkcd.com', 80))
except BlockingIOError:
    pass
```

Irritatingly, a non-blocking socket throws an exception from `connect`, even when it is working normally. This exception replicates the irritating behavior of the underlying C function, which sets `errno` to `EINPROGRESS` to tell you it has begun.

Now our crawler needs a way to know when the connection is established, so it can send the HTTP request. We could simply keep trying in a tight loop:

```
request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(url)
encoded = request.encode('ascii')

while True:
    try:
        sock.send(encoded)
        break  # Done.
    except OSError as e:
        pass

print('sent')
```

This method not only wastes electricity, but it cannot efficiently await events on *multiple* sockets. In ancient times, BSD Unix's solution to this problem was `select`, a C function that waits for an event to occur on a non-blocking socket or a small array of them. Nowadays the demand for Internet applications with huge numbers of connections has led to replacements like `poll`, then `kqueue` on BSD and `epoll` on Linux. These APIs are similar to `select`, but perform well with very large numbers of connections.

Python 3.4's `DefaultSelector` uses the best `select`-like function available on your system. To register for notifications about network I/O, we create a non-blocking socket and register it with the default selector:

```
from selectors import DefaultSelector, EVENT_WRITE

selector = DefaultSelector()

sock = socket.socket()
sock.setblocking(False)
try:
    sock.connect(('xkcd.com', 80))
except BlockingIOError:
    pass
```

```
def connected():
    selector.unregister(sock.fileno())
    print('connected!')

selector.register(sock.fileno(), EVENT_WRITE, connected)
```

We disregard the spurious error and call `selector.register`, passing in the socket's file descriptor and a constant that expresses what event we are waiting for. To be notified when the connection is established, we pass `EVENT_WRITE`: that is, we want to know when the socket is "writable". We also pass a Python function, `connected`, to run when that event occurs. Such a function is known as a *callback*.

We process I/O notifications as the selector receives them, in a loop:

```
def loop():
    while True:
        events = selector.select()
        for event_key, event_mask in events:
            callback = event_key.data
            callback()
```

The `connected` callback is stored as `event_key.data`, which we retrieve and execute once the non-blocking socket is connected.

Unlike in our fast-spinning loop above, the call to `select` here pauses, awaiting the next I/O events. Then the loop runs callbacks that are waiting for these events. Operations that have not completed remain pending until some future tick of the event loop.

What have we demonstrated already? We showed how to begin an operation and execute a callback when the operation is ready. An async *framework* builds on the two features we have shown—non-blocking sockets and the event loop—to run concurrent operations on a single thread.

We have achieved "concurrency" here, but not what is traditionally called "parallelism". That is, we built a tiny system that does overlapping I/O. It is capable of beginning new operations while others are in flight. It does not actually utilize multiple cores to execute computation in parallel. But then, this system is designed for I/O-bound problems, not CPU-bound ones.[4]

So our event loop is efficient at concurrent I/O because it does not devote thread resources to each connection. But before we proceed, it is important to correct a common misapprehension that async is *faster* than multithreading. Often it is not—indeed, in Python, an event loop like ours is moderately slower than multithreading at serving a small number of very active connections. In a runtime without a global interpreter lock, threads would perform even better on such a workload. What asynchronous I/O is right for, is applications with many slow or sleepy connections with infrequent events.[5][6]

[4] Python's global interpreter lock prohibits running Python code in parallel in one process anyway. Parallelizing CPU-bound algorithms in Python requires multiple processes, or writing the parallel portions of the code in C. But that is a topic for another day.

[5] Jesse listed indications and contraindications for using async in "What Is Async, How Does It Work, And When Should I Use It?", available at pyvideo.org.

[6] Mike Bayer compared the throughput of asyncio and multithreading for different workloads in his "Asynchronous Python and Databases": http://techspot.zzzeek.org/2015/02/15/asynchronous-python-and-databases/

5.5 Programming With Callbacks

With the runty async framework we have built so far, how can we build a web crawler? Even a simple URL-fetcher is painful to write.

We begin with global sets of the URLs we have yet to fetch, and the URLs we have seen:

```
urls_todo = set(['/'])
seen_urls = set(['/'])
```

The `seen_urls` set includes `urls_todo` plus completed URLs. The two sets are initialized with the root URL "/".

Fetching a page will require a series of callbacks. The `connected` callback fires when a socket is connected, and sends a GET request to the server. But then it must await a response, so it registers another callback. If, when that callback fires, it cannot read the full response yet, it registers again, and so on.

Let us collect these callbacks into a `Fetcher` object. It needs a URL, a socket object, and a place to accumulate the response bytes:

```
class Fetcher:
    def __init__(self, url):
        self.response = b''  # Empty array of bytes.
        self.url = url
        self.sock = None
```

We begin by calling `Fetcher.fetch`:

```
    # Method on Fetcher class.
    def fetch(self):
        self.sock = socket.socket()
        self.sock.setblocking(False)
        try:
            self.sock.connect(('xkcd.com', 80))
        except BlockingIOError:
            pass

        # Register next callback.
        selector.register(self.sock.fileno(),
                          EVENT_WRITE,
                          self.connected)
```

The `fetch` method begins connecting a socket. But notice the method returns before the connection is established. It must return control to the event loop to wait for the connection. To understand why, imagine our whole application was structured so:

```
# Begin fetching http://xkcd.com/353/
fetcher = Fetcher('/353/')
fetcher.fetch()

while True:
    events = selector.select()
    for event_key, event_mask in events:
        callback = event_key.data
        callback(event_key, event_mask)
```

All event notifications are processed in the event loop when it calls `select`. Hence `fetch` must hand control to the event loop, so that the program knows when the socket has connected. Only then does the loop run the `connected` callback, which was registered at the end of `fetch` above.

Here is the implementation of `connected`:

```
# Method on Fetcher class.
def connected(self, key, mask):
    print('connected!')
    selector.unregister(key.fd)
    request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(self.url)
    self.sock.send(request.encode('ascii'))

    # Register the next callback.
    selector.register(key.fd,
                      EVENT_READ,
                      self.read_response)
```

The method sends a GET request. A real application would check the return value of `send` in case the whole message cannot be sent at once. But our request is small and our application unsophisticated. It blithely calls `send`, then waits for a response. Of course, it must register yet another callback and relinquish control to the event loop. The next and final callback, `read_response`, processes the server's reply:

```
# Method on Fetcher class.
def read_response(self, key, mask):
    global stopped

    chunk = self.sock.recv(4096)  # 4k chunk size.
    if chunk:
        self.response += chunk
    else:
        selector.unregister(key.fd)  # Done reading.
        links = self.parse_links()

        # Python set-logic:
        for link in links.difference(seen_urls):
            urls_todo.add(link)
            Fetcher(link).fetch()  # <- New Fetcher.

        seen_urls.update(links)
        urls_todo.remove(self.url)
        if not urls_todo:
            stopped = True
```

The callback is executed each time the selector sees that the socket is "readable", which could mean two things: the socket has data or it is closed.

The callback asks for up to four kilobytes of data from the socket. If less is ready, `chunk` contains whatever data is available. If there is more, `chunk` is four kilobytes long and the socket remains readable, so the event loop runs this callback again on the next tick. When the response is complete, the server has closed the socket and `chunk` is empty.

The `parse_links` method, not shown, returns a set of URLs. We start a new fetcher for each new URL, with no concurrency cap. Note a nice feature of async programming with callbacks: we need no mutex around changes to shared data, such as when we add links to `seen_urls`. There is no preemptive multitasking, so we cannot be interrupted at arbitrary points in our code.

We add a global `stopped` variable and use it to control the loop:

```
stopped = False

def loop():
    while not stopped:
        events = selector.select()
        for event_key, event_mask in events:
            callback = event_key.data
            callback()
```

Once all pages are downloaded the fetcher stops the global event loop and the program exits.

This example makes async's problem plain: spaghetti code. We need some way to express a series of computations and I/O operations, and schedule multiple such series of operations to run concurrently. But without threads, a series of operations cannot be collected into a single function: whenever a function begins an I/O operation, it explicitly saves whatever state will be needed in the future, then returns. You are responsible for thinking about and writing this state-saving code.

Let us explain what we mean by that. Consider how simply we fetched a URL on a thread with a conventional blocking socket:

```
# Blocking version.
def fetch(url):
    sock = socket.socket()
    sock.connect(('xkcd.com', 80))
    request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(url)
    sock.send(request.encode('ascii'))
    response = b''
    chunk = sock.recv(4096)
    while chunk:
        response += chunk
        chunk = sock.recv(4096)

    # Page is now downloaded.
    links = parse_links(response)
    q.add(links)
```

What state does this function remember between one socket operation and the next? It has the socket, a URL, and the accumulating `response`. A function that runs on a thread uses basic features of the programming language to store this temporary state in local variables, on its stack. The function also has a "continuation"—that is, the code it plans to execute after I/O completes. The runtime remembers the continuation by storing the thread's instruction pointer. You need not think about restoring these local variables and the continuation after I/O. It is built in to the language.

But with a callback-based async framework, these language features are no help. While waiting for I/O, a function must save its state explicitly, because the function returns and loses its stack frame before I/O completes. In lieu of local variables, our callback-based example stores `sock` and `response` as attributes of `self`, the Fetcher instance. In lieu of the instruction pointer, it stores

its continuation by registering the callbacks `connected` and `read_response`. As the application's features grow, so does the complexity of the state we manually save across callbacks. Such onerous bookkeeping makes the coder prone to migraines.

Even worse, what happens if a callback throws an exception, before it schedules the next callback in the chain? Say we did a poor job on the `parse_links` method and it throws an exception parsing some HTML:

```
Traceback (most recent call last):
  File "loop-with-callbacks.py", line 111, in <module>
    loop()
  File "loop-with-callbacks.py", line 106, in loop
    callback(event_key, event_mask)
  File "loop-with-callbacks.py", line 51, in read_response
    links = self.parse_links()
  File "loop-with-callbacks.py", line 67, in parse_links
    raise Exception('parse error')
Exception: parse error
```

The stack trace shows only that the event loop was running a callback. We do not remember what led to the error. The chain is broken on both ends: we forgot where we were going and whence we came. This loss of context is called "stack ripping", and in many cases it confounds the investigator. Stack ripping also prevents us from installing an exception handler for a chain of callbacks, the way a "try / except" block wraps a function call and its tree of descendents.[7]

So, even apart from the long debate about the relative efficiencies of multithreading and async, there is this other debate regarding which is more error-prone: threads are susceptible to data races if you make a mistake synchronizing them, but callbacks are stubborn to debug due to stack ripping.

5.6 Coroutines

We entice you with a promise. It is possible to write asynchronous code that combines the efficiency of callbacks with the classic good looks of multithreaded programming. This combination is achieved with a pattern called "coroutines". Using Python 3.4's standard asyncio library, and a package called "aiohttp", fetching a URL in a coroutine is very direct[8]:

```
@asyncio.coroutine
def fetch(self, url):
    response = yield from self.session.get(url)
    body = yield from response.read()
```

It is also scalable. Compared to the 50k of memory per thread and the operating system's hard limits on threads, a Python coroutine takes barely 3k of memory on Jesse's system. Python can easily start hundreds of thousands of coroutines.

The concept of a coroutine, dating to the elder days of computer science, is simple: it is a subroutine that can be paused and resumed. Whereas threads are preemptively multitasked by

[7] For a complex solution to this problem, see `http://www.tornadoweb.org/en/stable/stack_context.html`

[8] The `@asyncio.coroutine` decorator is not magical. In fact, if it decorates a generator function and the `PYTHONASYNCIODEBUG` environment variable is not set, the decorator does practically nothing. It just sets an attribute, `_is_coroutine`, for the convenience of other parts of the framework. It is possible to use asyncio with bare generators not decorated with `@asyncio.coroutine` at all.

the operating system, coroutines multitask cooperatively: they choose when to pause, and which coroutine to run next.

There are many implementations of coroutines; even in Python there are several. The coroutines in the standard "asyncio" library in Python 3.4 are built upon generators, a Future class, and the "yield from" statement. Starting in Python 3.5, coroutines are a native feature of the language itself[9]; however, understanding coroutines as they were first implemented in Python 3.4, using pre-existing language facilities, is the foundation to tackle Python 3.5's native coroutines.

To explain Python 3.4's generator-based coroutines, we will engage in an exposition of generators and how they are used as coroutines in asyncio, and trust you will enjoy reading it as much as we enjoyed writing it. Once we have explained generator-based coroutines, we shall use them in our async web crawler.

5.7 How Python Generators Work

Before you grasp Python generators, you have to understand how regular Python functions work. Normally, when a Python function calls a subroutine, the subroutine retains control until it returns, or throws an exception. Then control returns to the caller:

```
>>> def foo():
...     bar()
...
>>> def bar():
...     pass
```

The standard Python interpreter is written in C. The C function that executes a Python function is called, mellifluously, `PyEval_EvalFrameEx`. It takes a Python stack frame object and evaluates Python bytecode in the context of the frame. Here is the bytecode for `foo`:

```
>>> import dis
>>> dis.dis(foo)
  2           0 LOAD_GLOBAL              0 (bar)
              3 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
              6 POP_TOP
              7 LOAD_CONST               0 (None)
             10 RETURN_VALUE
```

The `foo` function loads `bar` onto its stack and calls it, then pops its return value from the stack, loads `None` onto the stack, and returns `None`.

When `PyEval_EvalFrameEx` encounters the `CALL_FUNCTION` bytecode, it creates a new Python stack frame and recurses: that is, it calls `PyEval_EvalFrameEx` recursively with the new frame, which is used to execute `bar`.

It is crucial to understand that Python stack frames are allocated in heap memory! The Python interpreter is a normal C program, so its stack frames are normal stack frames. But the *Python* stack frames it manipulates are on the heap. Among other surprises, this means a Python stack frame can outlive its function call. To see this interactively, save the current frame from within `bar`:

[9] Python 3.5's built-in coroutines are described in PEP 492[10], "Coroutines with async and await syntax."

```
>>> import inspect
>>> frame = None
>>> def foo():
...     bar()
...
>>> def bar():
...     global frame
...     frame = inspect.currentframe()
...
>>> foo()
>>> # The frame was executing the code for 'bar'.
>>> frame.f_code.co_name
'bar'
>>> # Its back pointer refers to the frame for 'foo'.
>>> caller_frame = frame.f_back
>>> caller_frame.f_code.co_name
'foo'
```

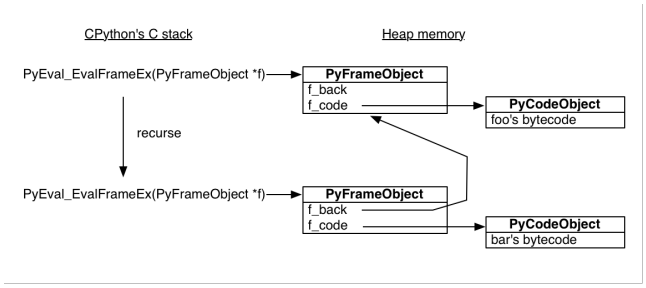

Figure 5.1: Function Calls

The stage is now set for Python generators, which use the same building blocks—code objects and stack frames—to marvelous effect.

This is a generator function:

```
>>> def gen_fn():
...     result = yield 1
...     print('result of yield: {}'.format(result))
...     result2 = yield 2
...     print('result of 2nd yield: {}'.format(result2))
...     return 'done'
...
```

When Python compiles `gen_fn` to bytecode, it sees the `yield` statement and knows that `gen_fn` is a generator function, not a regular one. It sets a flag to remember this fact:

```
>>> # The generator flag is bit position 5.
>>> generator_bit = 1 << 5
>>> bool(gen_fn.__code__.co_flags & generator_bit)
True
```

When you call a generator function, Python sees the generator flag, and it does not actually run the function. Instead, it creates a generator:

```
>>> gen = gen_fn()
>>> type(gen)
<class 'generator'>
```

A Python generator encapsulates a stack frame plus a reference to some code, the body of `gen_fn`:

```
>>> gen.gi_code.co_name
'gen_fn'
```

All generators from calls to `gen_fn` point to this same code. But each has its own stack frame. This stack frame is not on any actual stack, it sits in heap memory waiting to be used:

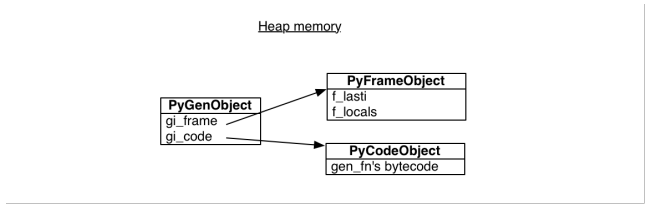

Figure 5.2: Generators

The frame has a "last instruction" pointer, the instruction it executed most recently. In the beginning, the last instruction pointer is -1, meaning the generator has not begun:

```
>>> gen.gi_frame.f_lasti
-1
```

When we call `send`, the generator reaches its first `yield`, and pauses. The return value of `send` is 1, since that is what `gen` passes to the `yield` expression:

```
>>> gen.send(None)
1
```

The generator's instruction pointer is now 3 bytecodes from the start, part way through the 56 bytes of compiled Python:

```
>>> gen.gi_frame.f_lasti
3
>>> len(gen.gi_code.co_code)
56
```

The generator can be resumed at any time, from any function, because its stack frame is not actually on the stack: it is on the heap. Its position in the call hierarchy is not fixed, and it need not obey the first-in, last-out order of execution that regular functions do. It is liberated, floating free like a cloud.

We can send the value "hello" into the generator and it becomes the result of the `yield` expression, and the generator continues until it yields 2:

```
>>> gen.send('hello')
result of yield: hello
2
```

Its stack frame now contains the local variable `result`:

```
>>> gen.gi_frame.f_locals
{'result': 'hello'}
```

Other generators created from `gen_fn` will have their own stack frames and local variables.

When we call `send` again, the generator continues from its second `yield`, and finishes by raising the special `StopIteration` exception:

```
>>> gen.send('goodbye')
result of 2nd yield: goodbye
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration: done
```

The exception has a value, which is the return value of the generator: the string `"done"`.

5.8 Building Coroutines With Generators

So a generator can pause, and it can be resumed with a value, and it has a return value. Sounds like a good primitive upon which to build an async programming model, without spaghetti callbacks! We want to build a "coroutine": a routine that is cooperatively scheduled with other routines in the program. Our coroutines will be a simplified version of those in Python's standard "asyncio" library. As in asyncio, we will use generators, futures, and the "yield from" statement.

First we need a way to represent some future result that a coroutine is waiting for. A stripped-down version:

```
class Future:
    def __init__(self):
        self.result = None
        self._callbacks = []

    def add_done_callback(self, fn):
        self._callbacks.append(fn)

    def set_result(self, result):
        self.result = result
        for fn in self._callbacks:
            fn(self)
```

A future is initially "pending". It is "resolved" by a call to `set_result`.[11]

Let us adapt our fetcher to use futures and coroutines. We wrote `fetch` with a callback:

```
class Fetcher:
    def fetch(self):
        self.sock = socket.socket()
        self.sock.setblocking(False)
        try:
```

[11] This future has many deficiencies. For example, once this future is resolved, a coroutine that yields it should resume immediately instead of pausing, but with our code it does not. See asyncio's Future class for a complete implementation.

```
            self.sock.connect(('xkcd.com', 80))
        except BlockingIOError:
            pass
        selector.register(self.sock.fileno(),
                          EVENT_WRITE,
                          self.connected)

    def connected(self, key, mask):
        print('connected!')
        # And so on....
```

The `fetch` method begins connecting a socket, then registers the callback, `connected`, to be executed when the socket is ready. Now we can combine these two steps into one coroutine:

```
    def fetch(self):
        sock = socket.socket()
        sock.setblocking(False)
        try:
            sock.connect(('xkcd.com', 80))
        except BlockingIOError:
            pass

        f = Future()

        def on_connected():
            f.set_result(None)

        selector.register(sock.fileno(),
                          EVENT_WRITE,
                          on_connected)
        yield f
        selector.unregister(sock.fileno())
        print('connected!')
```

Now `fetch` is a generator function, rather than a regular one, because it contains a `yield` statement. We create a pending future, then yield it to pause `fetch` until the socket is ready. The inner function `on_connected` resolves the future.

But when the future resolves, what resumes the generator? We need a coroutine *driver*. Let us call it "task":

```
class Task:
    def __init__(self, coro):
        self.coro = coro
        f = Future()
        f.set_result(None)
        self.step(f)

    def step(self, future):
        try:
            next_future = self.coro.send(future.result)
        except StopIteration:
```

```
            return

        next_future.add_done_callback(self.step)

# Begin fetching http://xkcd.com/353/
fetcher = Fetcher('/353/')
Task(fetcher.fetch())

loop()
```

The task starts the `fetch` generator by sending `None` into it. Then `fetch` runs until it yields a future, which the task captures as `next_future`. When the socket is connected, the event loop runs the callback `on_connected`, which resolves the future, which calls `step`, which resumes `fetch`.

5.9 Factoring Coroutines With `yield from`Factoring Coroutines With yield from

Once the socket is connected, we send the HTTP GET request and read the server response. These steps need no longer be scattered among callbacks; we gather them into the same generator function:

```
    def fetch(self):
        # ... connection logic from above, then:
        sock.send(request.encode('ascii'))

        while True:
            f = Future()

            def on_readable():
                f.set_result(sock.recv(4096))

            selector.register(sock.fileno(),
                              EVENT_READ,
                              on_readable)
            chunk = yield f
            selector.unregister(sock.fileno())
            if chunk:
                self.response += chunk
            else:
                # Done reading.
                break
```

This code, which reads a whole message from a socket, seems generally useful. How can we factor it from `fetch` into a subroutine? Now Python 3's celebrated `yield from` takes the stage. It lets one generator *delegate* to another.

To see how, let us return to our simple generator example:

```
>>> def gen_fn():
...     result = yield 1
...     print('result of yield: {}'.format(result))
```

```
...     result2 = yield 2
...     print('result of 2nd yield: {}'.format(result2))
...     return 'done'
...
```

To call this generator from another generator, delegate to it with `yield from`:

```
>>> # Generator function:
>>> def caller_fn():
...     gen = gen_fn()
...     rv = yield from gen
...     print('return value of yield-from: {}'
...           .format(rv))
...
>>> # Make a generator from the
>>> # generator function.
>>> caller = caller_fn()
```

The `caller` generator acts as if it were `gen`, the generator it is delegating to:

```
>>> caller.send(None)
1
>>> caller.gi_frame.f_lasti
15
>>> caller.send('hello')
result of yield: hello
2
>>> caller.gi_frame.f_lasti  # Hasn't advanced.
15
>>> caller.send('goodbye')
result of 2nd yield: goodbye
return value of yield-from: done
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

While `caller` yields from `gen`, `caller` does not advance. Notice that its instruction pointer remains at 15, the site of its `yield from` statement, even while the inner generator `gen` advances from one `yield` statement to the next.[12] From our perspective outside `caller`, we cannot tell if the values it yields are from `caller` or from the generator it delegates to. And from inside `gen`, we cannot tell if values are sent in from `caller` or from outside it. The `yield from` statement is a frictionless channel, through which values flow in and out of `gen` until `gen` completes.

A coroutine can delegate work to a sub-coroutine with `yield from` and receive the result of the work. Notice, above, that `caller` printed "return value of yield-from: done". When `gen` completed, its return value became the value of the `yield from` statement in `caller`:

```
    rv = yield from gen
```

[12] In fact, this is exactly how "yield from" works in CPython. A function increments its instruction pointer before executing each statement. But after the outer generator executes "yield from", it subtracts 1 from its instruction pointer to keep itself pinned at the "yield from" statement. Then it yields to *its* caller. The cycle repeats until the inner generator throws `StopIteration`, at which point the outer generator finally allows itself to advance to the next instruction.

Earlier, when we criticized callback-based async programming, our most strident complaint was about "stack ripping": when a callback throws an exception, the stack trace is typically useless. It only shows that the event loop was running the callback, not *why*. How do coroutines fare?

```
>>> def gen_fn():
...     raise Exception('my error')
>>> caller = caller_fn()
>>> caller.send(None)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 3, in caller_fn
  File "<input>", line 2, in gen_fn
Exception: my error
```

This is much more useful! The stack trace shows `caller_fn` was delegating to `gen_fn` when it threw the error. Even more comforting, we can wrap the call to a sub-coroutine in an exception handler, the same is with normal subroutines:

```
>>> def gen_fn():
...     yield 1
...     raise Exception('uh oh')
...
>>> def caller_fn():
...     try:
...         yield from gen_fn()
...     except Exception as exc:
...         print('caught {}'.format(exc))
...
>>> caller = caller_fn()
>>> caller.send(None)
1
>>> caller.send('hello')
caught uh oh
```

So we factor logic with sub-coroutines just like with regular subroutines. Let us factor some useful sub-coroutines from our fetcher. We write a `read` coroutine to receive one chunk:

```
def read(sock):
    f = Future()

    def on_readable():
        f.set_result(sock.recv(4096))

    selector.register(sock.fileno(), EVENT_READ, on_readable)
    chunk = yield f  # Read one chunk.
    selector.unregister(sock.fileno())
    return chunk
```

We build on `read` with a `read_all` coroutine that receives a whole message:

```
def read_all(sock):
    response = []
```

```
    # Read whole response.
    chunk = yield from read(sock)
    while chunk:
        response.append(chunk)
        chunk = yield from read(sock)

    return b''.join(response)
```

If you squint the right way, the `yield from` statements disappear and these look like conventional functions doing blocking I/O. But in fact, `read` and `read_all` are coroutines. Yielding from `read` pauses `read_all` until the I/O completes. While `read_all` is paused, asyncio's event loop does other work and awaits other I/O events; `read_all` is resumed with the result of `read` on the next loop tick once its event is ready.

At the stack's root, `fetch` calls `read_all`:

```
class Fetcher:
    def fetch(self):
         # ... connection logic from above, then:
        sock.send(request.encode('ascii'))
        self.response = yield from read_all(sock)
```

Miraculously, the Task class needs no modification. It drives the outer `fetch` coroutine just the same as before:

```
Task(fetcher.fetch())
loop()
```

When `read` yields a future, the task receives it through the channel of `yield from` statements, precisely as if the future were yielded directly from `fetch`. When the loop resolves a future, the task sends its result into `fetch`, and the value is received by `read`, exactly as if the task were driving `read` directly:

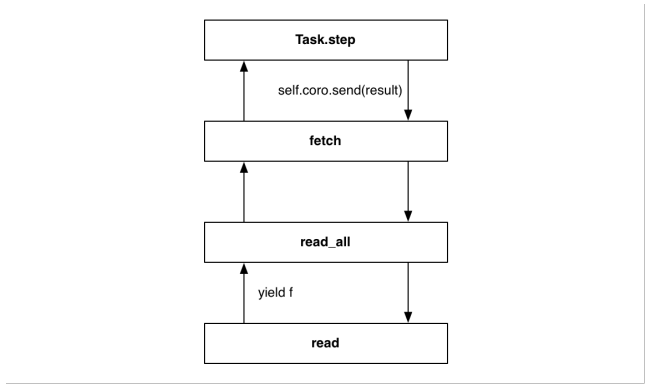

Figure 5.3: Yield From

To perfect our coroutine implementation, we polish out one mar: our code uses `yield` when it waits for a future, but `yield from` when it delegates to a sub-coroutine. It would be more refined if we used `yield from` whenever a coroutine pauses. Then a coroutine need not concern itself with what type of thing it awaits.

We take advantage of the deep correspondence in Python between generators and iterators. Advancing a generator is, to the caller, the same as advancing an iterator. So we make our Future class iterable by implementing a special method:

```
# Method on Future class.
def __iter__(self):
    # Tell Task to resume me here.
    yield self
    return self.result
```

The future's `__iter__` method is a coroutine that yields the future itself. Now when we replace code like this:

```
# f is a Future.
yield f
```

...with this:

```
# f is a Future.
yield from f
```

...the outcome is the same! The driving Task receives the future from its call to `send`, and when the future is resolved it sends the new result back into the coroutine.

What is the advantage of using `yield from` everywhere? Why is that better than waiting for futures with `yield` and delegating to sub-coroutines with `yield from`? It is better because now, a method can freely change its implementation without affecting the caller: it might be a normal method that returns a future that will *resolve* to a value, or it might be a coroutine that contains `yield from` statements and *returns* a value. In either case, the caller need only `yield from` the method in order to wait for the result.

Gentle reader, we have reached the end of our enjoyable exposition of coroutines in asyncio. We peered into the machinery of generators, and sketched an implementation of futures and tasks. We outlined how asyncio attains the best of both worlds: concurrent I/O that is more efficient than threads and more legible than callbacks. Of course, the real asyncio is much more sophisticated than our sketch. The real framework addresses zero-copy I/O, fair scheduling, exception handling, and an abundance of other features.

To an asyncio user, coding with coroutines is much simpler than you saw here. In the code above we implemented coroutines from first principles, so you saw callbacks, tasks, and futures. You even saw non-blocking sockets and the call to `select`. But when it comes time to build an application with asyncio, none of this appears in your code. As we promised, you can now sleekly fetch a URL:

```
@asyncio.coroutine
def fetch(self, url):
    response = yield from self.session.get(url)
    body = yield from response.read()
```

Satisfied with this exposition, we return to our original assignment: to write an async web crawler, using asyncio.

5.10 Coordinating Coroutines

We began by describing how we want our crawler to work. Now it is time to implement it with asyncio coroutines.

Our crawler will fetch the first page, parse its links, and add them to a queue. After this it fans out across the website, fetching pages concurrently. But to limit load on the client and server, we want some maximum number of workers to run, and no more. Whenever a worker finishes fetching a page, it should immediately pull the next link from the queue. We will pass through periods when there is not enough work to go around, so some workers must pause. But when a worker hits a page rich with new links, then the queue suddenly grows and any paused workers should wake and get cracking. Finally, our program must quit once its work is done.

Imagine if the workers were threads. How would we express the crawler's algorithm? We could use a synchronized queue[13] from the Python standard library. Each time an item is put in the queue, the queue increments its count of "tasks". Worker threads call `task_done` after completing work on an item. The main thread blocks on `Queue.join` until each item put in the queue is matched by a `task_done` call, then it exits.

Coroutines use the exact same pattern with an asyncio queue! First we import it[14]:

```
try:
    from asyncio import JoinableQueue as Queue
except ImportError:
    # In Python 3.5, asyncio.JoinableQueue is
    # merged into Queue.
    from asyncio import Queue
```

We collect the workers' shared state in a crawler class, and write the main logic in its `crawl` method. We start `crawl` on a coroutine and run asyncio's event loop until `crawl` finishes:

```
loop = asyncio.get_event_loop()

crawler = crawling.Crawler('http://xkcd.com',
                           max_redirect=10)

loop.run_until_complete(crawler.crawl())
```

The crawler begins with a root URL and `max_redirect`, the number of redirects it is willing to follow to fetch any one URL. It puts the pair `(URL, max_redirect)` in the queue. (For the reason why, stay tuned.)

```
class Crawler:
    def __init__(self, root_url, max_redirect):
        self.max_tasks = 10
        self.max_redirect = max_redirect
        self.q = Queue()
        self.seen_urls = set()

        # aiohttp's ClientSession does connection pooling and
        # HTTP keep-alives for us.
```

[13] https://docs.python.org/3/library/queue.html
[14] https://docs.python.org/3/library/asyncio-sync.html

```
        self.session = aiohttp.ClientSession(loop=loop)

        # Put (URL, max_redirect) in the queue.
        self.q.put((root_url, self.max_redirect))
```

The number of unfinished tasks in the queue is now one. Back in our main script, we launch the event loop and the `crawl` method:

```
loop.run_until_complete(crawler.crawl())
```

The `crawl` coroutine kicks off the workers. It is like a main thread: it blocks on `join` until all tasks are finished, while the workers run in the background.

```
    @asyncio.coroutine
    def crawl(self):
        """Run the crawler until all work is done."""
        workers = [asyncio.Task(self.work())
                   for _ in range(self.max_tasks)]

        # When all work is done, exit.
        yield from self.q.join()
        for w in workers:
            w.cancel()
```

If the workers were threads we might not wish to start them all at once. To avoid creating expensive threads until it is certain they are necessary, a thread pool typically grows on demand. But coroutines are cheap, so we simply start the maximum number allowed.

It is interesting to note how we shut down the crawler. When the `join` future resolves, the worker tasks are alive but suspended: they wait for more URLs but none come. So, the main coroutine cancels them before exiting. Otherwise, as the Python interpreter shuts down and calls all objects' destructors, living tasks cry out:

```
ERROR:asyncio:Task was destroyed but it is pending!
```

And how does `cancel` work? Generators have a feature we have not yet shown you. You can throw an exception into a generator from outside:

```
>>> gen = gen_fn()
>>> gen.send(None)  # Start the generator as usual.
1
>>> gen.throw(Exception('error'))
Traceback (most recent call last):
  File "<input>", line 3, in <module>
  File "<input>", line 2, in gen_fn
Exception: error
```

The generator is resumed by `throw`, but it is now raising an exception. If no code in the generator's call stack catches it, the exception bubbles back up to the top. So to cancel a task's coroutine:

```
    # Method of Task class.
    def cancel(self):
        self.coro.throw(CancelledError)
```

Wherever the generator is paused, at some `yield from` statement, it resumes and throws an exception. We handle cancellation in the task's `step` method:

```
# Method of Task class.
def step(self, future):
    try:
        next_future = self.coro.send(future.result)
    except CancelledError:
        self.cancelled = True
        return
    except StopIteration:
        return

    next_future.add_done_callback(self.step)
```

Now the task knows it is cancelled, so when it is destroyed it does not rage against the dying of the light.

Once `crawl` has canceled the workers, it exits. The event loop sees that the coroutine is complete (we shall see how later), and it too exits:

```
loop.run_until_complete(crawler.crawl())
```

The `crawl` method comprises all that our main coroutine must do. It is the worker coroutines that get URLs from the queue, fetch them, and parse them for new links. Each worker runs the `work` coroutine independently:

```
@asyncio.coroutine
def work(self):
    while True:
        url, max_redirect = yield from self.q.get()

        # Download page and add new links to self.q.
        yield from self.fetch(url, max_redirect)
        self.q.task_done()
```

Python sees that this code contains `yield from` statements, and compiles it into a generator function. So in `crawl`, when the main coroutine calls `self.work` ten times, it does not actually execute this method: it only creates ten generator objects with references to this code. It wraps each in a Task. The Task receives each future the generator yields, and drives the generator by calling `send` with each future's result when the future resolves. Because the generators have their own stack frames, they run independently, with separate local variables and instruction pointers.

The worker coordinates with its fellows via the queue. It waits for new URLs with:

```
url, max_redirect = yield from self.q.get()
```

The queue's `get` method is itself a coroutine: it pauses until someone puts an item in the queue, then resumes and returns the item.

Incidentally, this is where the worker will be paused at the end of the crawl, when the main coroutine cancels it. From the coroutine's perspective, its last trip around the loop ends when `yield from` raises a `CancelledError`.

When a worker fetches a page it parses the links and puts new ones in the queue, then calls `task_done` to decrement the counter. Eventually, a worker fetches a page whose URLs have all been fetched already, and there is also no work left in the queue. Thus this worker's call to `task_done` decrements the counter to zero. Then `crawl`, which is waiting for the queue's `join` method, is unpaused and finishes.

We promised to explain why the items in the queue are pairs, like:

```
# URL to fetch, and the number of redirects left.
('http://xkcd.com/353', 10)
```

New URLs have ten redirects remaining. Fetching this particular URL results in a redirect to a new location with a trailing slash. We decrement the number of redirects remaining, and put the next location in the queue:

```
# URL with a trailing slash. Nine redirects left.
('http://xkcd.com/353/', 9)
```

The `aiohttp` package we use would follow redirects by default and give us the final response. We tell it not to, however, and handle redirects in the crawler, so it can coalesce redirect paths that lead to the same destination: if we have already seen this URL, it is in `self.seen_urls` and we have already started on this path from a different entry point:

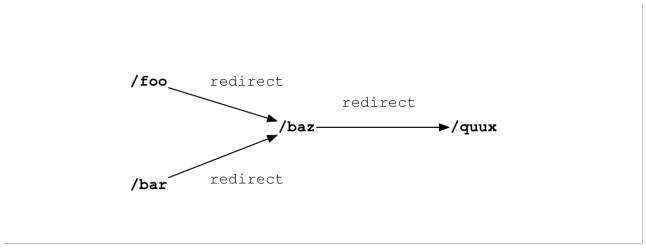

Figure 5.4: Redirects

The crawler fetches "foo" and sees it redirects to "baz", so it adds "baz" to the queue and to `seen_urls`. If the next page it fetches is "bar", which also redirects to "baz", the fetcher does not enqueue "baz" again. If the response is a page, rather than a redirect, `fetch` parses it for links and puts new ones in the queue.

```
    @asyncio.coroutine
    def fetch(self, url, max_redirect):
        # Handle redirects ourselves.
        response = yield from self.session.get(
            url, allow_redirects=False)

        try:
            if is_redirect(response):
                if max_redirect > 0:
                    next_url = response.headers['location']
                    if next_url in self.seen_urls:
                        # We have been down this path before.
                        return
```

```
                # Remember we have seen this URL.
                self.seen_urls.add(next_url)

                # Follow the redirect. One less redirect remains.
                self.q.put_nowait((next_url, max_redirect - 1))
            else:
                links = yield from self.parse_links(response)
                # Python set-logic:
                for link in links.difference(self.seen_urls):
                    self.q.put_nowait((link, self.max_redirect))
                self.seen_urls.update(links)
        finally:
            # Return connection to pool.
            yield from response.release()
```

If this were multithreaded code, it would be lousy with race conditions. For example, the worker checks if a link is in `seen_urls`, and if not the worker puts it in the queue and adds it to `seen_urls`. If it were interrupted between the two operations, then another worker might parse the same link from a different page, also observe that it is not in `seen_urls`, and also add it to the queue. Now that same link is in the queue twice, leading (at best) to duplicated work and wrong statistics.

However, a coroutine is only vulnerable to interruption at `yield from` statements. This is a key difference that makes coroutine code far less prone to races than multithreaded code: multithreaded code must enter a critical section explicitly, by grabbing a lock, otherwise it is interruptible. A Python coroutine is uninterruptible by default, and only cedes control when it explicitly yields.

We no longer need a fetcher class like we had in the callback-based program. That class was a workaround for a deficiency of callbacks: they need some place to store state while waiting for I/O, since their local variables are not preserved across calls. But the `fetch` coroutine can store its state in local variables like a regular function does, so there is no more need for a class.

When `fetch` finishes processing the server response it returns to the caller, `work`. The `work` method calls `task_done` on the queue and then gets the next URL from the queue to be fetched.

When `fetch` puts new links in the queue it increments the count of unfinished tasks and keeps the main coroutine, which is waiting for `q.join`, paused. If, however, there are no unseen links and this was the last URL in the queue, then when `work` calls `task_done` the count of unfinished tasks falls to zero. That event unpauses `join` and the main coroutine completes.

The queue code that coordinates the workers and the main coroutine is like this[15]:

```
class Queue:
    def __init__(self):
        self._join_future = Future()
        self._unfinished_tasks = 0
        # ... other initialization ...

    def put_nowait(self, item):
        self._unfinished_tasks += 1
        # ... store the item ...
```

[15]The actual `asyncio.Queue` implementation uses an `asyncio.Event` in place of the Future shown here. The difference is an Event can be reset, whereas a Future cannot transition from resolved back to pending.

```
    def task_done(self):
        self._unfinished_tasks -= 1
        if self._unfinished_tasks == 0:
            self._join_future.set_result(None)

    @asyncio.coroutine
    def join(self):
        if self._unfinished_tasks > 0:
            yield from self._join_future
```

The main coroutine, `crawl`, yields from `join`. So when the last worker decrements the count of unfinished tasks to zero, it signals `crawl` to resume, and finish.

The ride is almost over. Our program began with the call to `crawl`:

```
loop.run_until_complete(self.crawler.crawl())
```

How does the program end? Since `crawl` is a generator function, calling it returns a generator. To drive the generator, asyncio wraps it in a task:

```
class EventLoop:
    def run_until_complete(self, coro):
        """Run until the coroutine is done."""
        task = Task(coro)
        task.add_done_callback(stop_callback)
        try:
            self.run_forever()
        except StopError:
            pass

class StopError(BaseException):
    """Raised to stop the event loop."""

def stop_callback(future):
    raise StopError
```

When the task completes, it raises `StopError`, which the loop uses as a signal that it has arrived at normal completion.

But what's this? The task has methods called `add_done_callback` and `result`? You might think that a task resembles a future. Your instinct is correct. We must admit a detail about the Task class we hid from you: a task is a future.

```
class Task(Future):
    """A coroutine wrapped in a Future."""
```

Normally a future is resolved by someone else calling `set_result` on it. But a task resolves *itself* when its coroutine stops. Remember from our earlier exploration of Python generators that when a generator returns, it throws the special `StopIteration` exception:

```
    # Method of class Task.
    def step(self, future):
        try:
```

```
            next_future = self.coro.send(future.result)
        except CancelledError:
            self.cancelled = True
            return
        except StopIteration as exc:

            # Task resolves itself with coro's return
            # value.
            self.set_result(exc.value)
            return

        next_future.add_done_callback(self.step)
```

So when the event loop calls `task.add_done_callback(stop_callback)`, it prepares to be stopped by the task. Here is `run_until_complete` again:

```
    # Method of event loop.
    def run_until_complete(self, coro):
        task = Task(coro)
        task.add_done_callback(stop_callback)
        try:
            self.run_forever()
        except StopError:
            pass
```

When the task catches `StopIteration` and resolves itself, the callback raises `StopError` from within the loop. The loop stops and the call stack is unwound to `run_until_complete`. Our program is finished.

5.11 Conclusion

Increasingly often, modern programs are I/O-bound instead of CPU-bound. For such programs, Python threads are the worst of both worlds: the global interpreter lock prevents them from actually executing computations in parallel, and preemptive switching makes them prone to races. Async is often the right pattern. But as callback-based async code grows, it tends to become a dishevelled mess. Coroutines are a tidy alternative. They factor naturally into subroutines, with sane exception handling and stack traces.

If we squint so that the `yield from` statements blur, a coroutine looks like a thread doing traditional blocking I/O. We can even coordinate coroutines with classic patterns from multi-threaded programming. There is no need for reinvention. Thus, compared to callbacks, coroutines are an inviting idiom to the coder experienced with multithreading.

But when we open our eyes and focus on the `yield from` statements, we see they mark points when the coroutine cedes control and allows others to run. Unlike threads, coroutines display where our code can be interrupted and where it cannot. In his illuminating essay "Unyielding"[16], Glyph Lefkowitz writes, "Threads make local reasoning difficult, and local reasoning is perhaps the most important thing in software development." Explicitly yielding, however, makes it possible to "understand the behavior (and thereby, the correctness) of a routine by examining the routine itself rather than examining the entire system."

[16] `https://glyph.twistedmatrix.com/2014/02/unyielding.html`

This chapter was written during a renaissance in the history of Python and async. Generator-based coroutines, whose devising you have just learned, were released in the "asyncio" module with Python 3.4 in March 2014. In September 2015, Python 3.5 was released with coroutines built in to the language itself. These native coroutinesare declared with the new syntax "async def", and instead of "yield from", they use the new "await" keyword to delegate to a coroutine or wait for a Future.

Despite these advances, the core ideas remain. Python's new native coroutines will be syntactically distinct from generators but work very similarly; indeed, they will share an implementation within the Python interpreter. Task, Future, and the event loop will continue to play their roles in asyncio.

Now that you know how asyncio coroutines work, you can largely forget the details. The machinery is tucked behind a dapper interface. But your grasp of the fundamentals empowers you to code correctly and efficiently in modern async environments.

An Event-Driven Web Framework

Leo Zovic

In 2013, I decided to write a web-based game prototyping tool[1] for card and board games called *House*. In these types of games, it is common for one player to wait for another player to make a move; however, when the other player finally does take action, we would like for the waiting player to be notified of the move quickly thereafter.

This is a problem that turns out to be more complicated than it first seems. In this chapter, we'll explore the issues with using HTTP to build this sort of interaction, and then we'll build a *web framework* in Common Lisp that allows us to solve similar problems in the future.

8.1 The Basics of HTTP Servers

At the simplest level, an HTTP exchange is a single request followed by a single response. A *client* sends a request, which includes a resource identifier, an HTTP version tag, some headers and some parameters. The *server* parses that request, figures out what to do about it, and sends a response which includes the same HTTP version tag, a response code, some headers and a response body.

Notice that, in this description, the server responds to a request from a specific client. In our case, we want each player to be updated about *any* moves as soon as they happen, rather than only getting notifications when their own move is made. This means we need the server to *push* messages to clients without first receiving a request for the information.[2]

There are several standard approaches to enabling server push over HTTP.

Comet/Long Poll

The "long poll" technique has the client send the server a new request as soon as it receives a response. Instead of fulfilling that request right away, the server waits on a subsequent event to respond. This is a bit of a semantic distinction, since the client is still taking action on every update.

[1] `https://github.com/Inaimathi/deal`

[2] One solution to this problem is to force the clients to *poll* the server. That is, each client would periodically send the server a request asking if anything has changed. This can work for simple applications, but in this chapter we're going to focus on the solutions available to you when this model stops working.

Server-Sent Events (SSE)

Server-sent events require that the client initiates a connection and then keeps it open. The server periodically writes new data to the connection without closing it, and the client interprets incoming new messages as they arrive rather than waiting for the response connection to terminate. This is a bit more efficient than the Comet/long poll approach because each message doesn't have to incur the overhead of new HTTP headers.

WebSockets

WebSockets are a communication protocol built on top of HTTP. The server and client open up an HTTP conversation, then perform a handshake and protocol escalation. The end result is that they're still communicating over TCP/IP, but they're not using HTTP to do it at all. The advantage this has over SSEs is that you can customize the protocol for efficiency.

Long-Lived Connections

These three approaches are quite different from one another, but they all share an important characteristic: they all depend on long-lived connections. Long polling depends on the server keeping requests around until new data is available, SSEs keep an open stream between client and server to which data is periodically written, and WebSockets change the protocol a particular connection is using, but leave it open.

To see why this might cause problems for your average HTTP server, let's consider how the underlying implementation might work.

Traditional HTTP Server Architecture

A single HTTP server processes many requests concurrently. Historically, many HTTP servers have used a *thread-per-request* architecture. That is, for each incoming request, the server creates a thread to do the work necessary to respond.

Since each of these connections is intended to be short-lived, we don't need many threads executing in parallel to handle them all. This model also simplifies the *implementation* of the server by enabling the server programmer to write code as if there were only one connection being handled at any given time. It also gives us the freedom to clean up failed or "zombie" connections and their associated resources by killing the corresponding thread and letting the garbage collector do its job.

The key observation is that an HTTP server hosting a "traditional" web application that has N concurrent users might only need to handle a very small fraction of N requests *in parallel* to succeed. For the type of interactive application that we are trying to build, N users will almost certainly require the application to maintain at least N connections in parallel, at once.

The consequence of keeping long-lived connections around is that we'll need either:

- A platform where threads are "cheap" enough that we can use large numbers of them at once.
- A server architecture that can handle many connections with a single thread.

There are programming environments such as Racket[3], Erlang[4], and Haskell[5] that provide thread-like constructs that are "lightweight" enough to consider the first option. This approach requires

[3] `http://racket-lang.org/`
[4] `http://www.erlang.org/`
[5] `http://hackage.haskell.org/package/base-4.7.0.1/docs/Control-Concurrent.html`

the programmer to explicitly deal with synchronization issues, which are going to be much more prevalent in a system where connections are open for a long time and likely all competing for similar resources. Specifically, if we have some sort of central data shared by several users simultaneously, we will need to coordinate reads and writes of that data in some way.

If we don't have cheap threads at our disposal or we are unwilling to work with explicit synchronization, we must consider having a single thread handle many connections.[6] In this model, our single thread is going to be handling tiny "slices" of many requests all at once, switching between them as efficiently as it possibly can. This system architecture pattern is most commonly referred to as *event-driven* or *event-based*.[7]

Since we are only managing a single thread, we don't have to worry as much about protecting shared resources from simultaneous access. However, we do have a unique problem of our own in this model. Since our single thread is working on all in-flight requests at once, we must make sure that it **never blocks**. Blocking on any connection blocks the entire server from making progress on any other request. We have to be able to move on to another client if the current one can't be serviced further, and we need to be able to do so in a manner that doesn't throw out the work done so far.[8]

While it is uncommon for a programmer to explicitly tell a thread to stop working, many common operations carry a risk of blocking. Because threads are so prevalent, and reasoning about asynchronicity is a heavy burden on the programmer, many languages and their frameworks assume that blocking on I/O is a desirable property. This makes it very easy to block somewhere *by accident*. Luckily, Common Lisp does provide us with a minimal set of asynchronous I/O primitives which we can build on top of.

Architectural Decisions

Now that we've studied the background of this problem, we've arrived at the point where we need to make informed decisions about *what* we are building.

At the time I started thinking about this project, Common Lisp didn't have a complete green-thread implementation, and the standard portable threading library[9] doesn't qualify as "really REALLY cheap". The options amounted to either picking a different language, or building an event-driven web server for my purpose. I chose the latter.

In addition to the server architecture, we also need to choose which of the three server-push approaches to use. The use-case we are considering (an interactive multiplayer board game) requires frequent updates to each client, but relatively sparse requests *from* each client, which fits the SSE approach to pushing updates, so we'll go with this.

Now that we've motivated our architectural decision and decided on a mechanism for simulating bidirectional communication between clients and server, let's get started on building our web framework. We'll start by building a relatively "dumb" server first, and then we'll extend it into a

[6] We could consider a more general system that handles N concurrent users with M threads for some configurable value of M; in this model, the N connections are said to be *multiplexed* across the M threads. In this chapter, we are going to focus on writing a program where M is fixed at 1; however, the lessons learned here should be partially applicable to the more general model.

[7] This nomenclature is a bit confusing, and has its origin in early operating-systems research. It refers to how communication is done between multiple concurrent processes. In a thread-based system, communication is done through a synchronized resource such as shared memory. In an event-based system, processes generally communicate through a queue where they post items that describe what they have done or what they want done, which is maintained by our single thread of execution. Since these items generally describe desired or past actions, they are referred to as 'events'.

[8] See Chapter 5 for another take on this problem.

[9] `http://common-lisp.net/project/bordeaux-threads/`

web-application framework that lets us focus on *what* our heavily-interactive program needs to do, and not *how* it is doing it.

8.2 Building an Event-Driven Web Server

Most programs that use a single process to manage concurrent streams of work use a pattern called an *event loop*. Let's look at what an event loop for our web server might look like.

The Event Loop

Our event loop needs to:

- listen for incoming connections;
- handle all new handshakes or incoming data on existing connections;
- clean up dangling sockets that are unexpectedly killed (e.g. by an interrupt)

```
(defmethod start ((port integer))
  (let ((server (socket-listen
         usocket:*wildcard-host* port
         :reuse-address t
         :element-type 'octet))
    (conns (make-hash-table)))
    (unwind-protect
     (loop (loop for ready
          in (wait-for-input
              (cons server (alexandria:hash-table-keys conns))
              :ready-only t)
          do (process-ready ready conns)))
      (loop for c being the hash-keys of conns
     do (loop while (socket-close c)))
      (loop while (socket-close server)))))
```

If you haven't written a Common Lisp program before, this code block requires some explanation. What we have written here is a *method definition*. While Lisp is popularly known as a functional language, it also has its own system for object-oriented programming called "The Common Lisp Object System", which is usually abbreviated as "CLOS".[10]

CLOS and Generic Functions

In CLOS, instead of focusing on classes and methods, we write *generic functions*[11] that are implemented as collections of *methods*. In this model, methods don't *belong to* classes, they *specialize on* types.[12] The `start` method we just wrote is a unary method where the argument `port` is *specialized on* the type `integer`. This means that we could have several implementations of `start` where `port` varies in type, and the runtime will select which implementation to use depending on the type of `port` when `start` is called.

[10] Pronounced "kloss", "see-loss" or "see-lows", depending on who you talk to.

[11] `http://www.gigamonkeys.com/book/object-reorientation-generic-functions.html`

[12] The Julia programming language takes a similar approach to object-oriented programming; you can learn more about it in Chapter 20.

More generally, methods can specialize on more than one argument. When a `method` is called, the runtime:

- dispatches on the type of its arguments to figure out which method body should be run, and
- runs the appropriate function.

Processing Sockets

We'll see another generic function at work in `process-ready`, which was called earlier from our event loop. It processes a ready socket with one of two methods, depending on the type of socket we are handling.

The two types we're concerned with are the `stream-usocket`, which represents a client socket that will make a request and expect to be sent some data back, and the `stream-server-usocket`, which represents our local TCP listener that will have new client connections for us to deal with.

If a `stream-server-socket` is `ready`, that means there's a new client socket waiting to start a conversation. We call `socket-accept` to accept the connection, and then put the result in our connection table so that our event loop can begin processing it with the others.

```
(defmethod process-ready ((ready stream-server-usocket) (conns hash-table))
  (setf (gethash (socket-accept ready :element-type 'octet) conns) nil))
```

When a `stream-usocket` is `ready`, that means that it has some bytes ready for us to read. (It's also possible that the other party has terminated the connection.)

```
(defmethod process-ready ((ready stream-usocket) (conns hash-table))
  (let ((buf (or (gethash ready conns)
         (setf (gethash ready conns)
               (make-instance 'buffer :bi-stream (flex-stream ready))))))
    (if (eq :eof (buffer! buf))
    (ignore-errors
      (remhash ready conns)
      (socket-close ready))
    (let ((too-big?
           (> (total-buffered buf)
          +max-request-size+))
          (too-old?
           (> (- (get-universal-time) (started buf))
          +max-request-age+))
          (too-needy?
           (> (tries buf)
          +max-buffer-tries+)))
      (cond (too-big?
         (error! +413+ ready)
         (remhash ready conns))
        ((or too-old? too-needy?)
         (error! +400+ ready)
         (remhash ready conns))
        ((and (request buf) (zerop (expecting buf)))
         (remhash ready conns)
         (when (contents buf)
           (setf (parameters (request buf))
```

```
          (nconc (parse buf) (parameters (request buf)))))
       (handler-case
           (handle-request ready (request buf))
         (http-assertion-error () (error! +400+ ready))
         ((and (not warning)
           (not simple-error)) (e)
           (error! +500+ ready e))))
      (t
       (setf (contents buf) nil)))))))
```

This is more involved than the first case. We:

1. Get the buffer associated with this socket, or create it if it doesn't exist yet;
2. Read output into that buffer, which happens in the call to `buffer!`;
3. If that read got us an `:eof`, the other side hung up, so we discard the socket *and* its buffer;
4. Otherwise, we check if the buffer is one of `complete?`, `too-big?`, `too-old?` or `too-needy?`. If so, we remove it from the connections table and return the appropriate HTTP response.

This is the first time we're seeing I/O in our event loop. In our discussion in Section 8.1, we mentioned that we have to be very careful about I/O in an event-driven system, because we could accidentally block our single thread. So, what do we do here to ensure that this doesn't happen? We have to explore our implementation of `buffer!` to find out exactly how this works.

Processing Connections Without Blocking

The basis of our approach to processing connections without blocking is the library function `read-char-no-hang`[13], which immediately returns `nil` when called on a stream that has no available data. Where there is data to be read, we use a buffer to store intermediate input for this connection.

```
(defmethod buffer! ((buffer buffer))
  (handler-case
      (let ((stream (bi-stream buffer)))
        (incf (tries buffer))
        (loop for char = (read-char-no-hang stream) until (null char)
           do (push char (contents buffer))
           do (incf (total-buffered buffer))
           when (request buffer) do (decf (expecting buffer))
           when (line-terminated? (contents buffer))
           do (multiple-value-bind (parsed expecting) (parse buffer)
            (setf (request buffer) parsed
                  (expecting buffer) expecting)
            (return char))
           when (> (total-buffered buffer) +max-request-size+) return char
           finally (return char)))
    (error () :eof)))
```

When `buffer!` is called on a `buffer`, it:

- increments the `tries` count, so that we can evict "needy" buffers in `process-ready`;
- loops to read characters from the input stream, and

[13] `http://clhs.lisp.se/Body/f_rd_c_1.htm`

- returns the last character it read if it has read all of the available input.

It also tracks any `\r\n\r\n` sequences so that we can later detect complete requests. Finally, if any error results, it returns an `:eof` to signal that `process-ready` should discard this connection.

The `buffer` type is a CLOS *class*[14]. Classes in CLOS let us define a type with fields called `slots`. We don't see the behaviours associated with `buffer` on the class definition, because (as we've already learned), we do that using generic functions like `buffer!`.

`defclass` does allow us to specify getters/setters (`readers`/`accessors`), and slot initializers; `:initform` specifies a default value, while `:initarg` identifies a hook that the caller of `make-instance` can use to provide a default value.

```
(defclass buffer ()
  ((tries :accessor tries :initform 0)
   (contents :accessor contents :initform nil)
   (bi-stream :reader bi-stream :initarg :bi-stream)
   (total-buffered :accessor total-buffered :initform 0)
   (started :reader started :initform (get-universal-time))
   (request :accessor request :initform nil)
   (expecting :accessor expecting :initform 0)))
```

Our `buffer` class has seven slots:

- `tries`, which keeps count of how many times we've tried reading into this buffer
- `contents`, which contains what we've read so far
- `bi-stream`, which a hack around some of those Common Lisp-specific, non-blocking-I/O annoyances I mentioned earlier
- `total-buffered`, which is a count of chars we've read so far
- `started`, which is a timestamp that tells us when we created this buffer
- `request`, which will eventually contain the request we construct from buffered data
- `expecting`, which will signal how many more chars we're expecting (if any) after we buffer the request headers

Interpreting Requests

Now that we've seen how we incrementally assemble full requests from bits of data that are pooled into our buffers, what happens when we have a full request ready for handling? This happens in the method `handle-request`.

```
(defmethod handle-request ((socket usocket) (req request))
  (aif (lookup (resource req) *handlers*)
       (funcall it socket (parameters req))
       (error! +404+ socket)))
```

This method adds another layer of error handling so that if the request is old, big, or needy, we can send a `400` response to indicate that the client provided us with some bad or slow data. However, if any *other* error happens here, it's because the programer made a mistake defining a *handler*, which should be treated as a `500` error. This will inform the client that something went wrong on the server as a result of their legitimate request.

[14] `http://www.gigamonkeys.com/book/object-reorientation-classes.html`

If the request is well-formed, we do the tiny and obvious job of looking up the requested resource in the `*handlers*` table. If we find one, we `funcall` it, passing along the client `socket` as well as the parsed request parameters. If there's no matching handler in the `*handlers*` table, we instead send along a `404` error. The handler system will be part of our full-fledged *web framework*, which we'll discuss in a later section.

We still haven't seen how requests are parsed and interpreted from one of our buffers, though. Let's look at that next:

```
(defmethod parse ((buf buffer))
  (let ((str (coerce (reverse (contents buf)) 'string)))
    (if (request buf)
        (parse-params str)
        (parse str))))
```

This high-level method delegates to a specialization of `parse` that works with plain strings, or to `parse-params` that interprets the buffer contents as HTTP parameters. These are called depending on how much of the request we've already processed; the final `parse` happens when we already have a partial `request` saved in the `buffer`, at which point we're only looking to parse the request body.

```
(defmethod parse ((str string))
  (let ((lines (split "\\r?\\n" str)))
    (destructuring-bind (req-type path http-version) (split " " (pop lines))
      (declare (ignore req-type))
      (assert-http (string= http-version "HTTP/1.1"))
      (let* ((path-pieces (split "\\?" path))
         (resource (first path-pieces))
         (parameters (second path-pieces))
         (req (make-instance 'request :resource resource)))
    (loop
       for header = (pop lines)
       for (name value) = (split ": " header)
       until (null name)
       do (push (cons (->keyword name) value) (headers req)))
    (setf (parameters req) (parse-params parameters))
    req))))

(defmethod parse-params ((params null)) nil)

(defmethod parse-params ((params string))
  (loop for pair in (split "&" params)
     for (name val) = (split "=" pair)
     collect (cons (->keyword name) (or val ""))))
```

In the `parse` method specializing on `string`, we transform the content into usable pieces. We do so on strings instead of working directly with buffers because this makes it easier to test the actual parsing code in an environment like an interpreter or REPL.

The parsing process is:

1. Split on `"\\r?\\n"`.
2. Split the first line of that on `" "` to get the request type (`POST`, `GET`, etc)/URI path/http-version.
3. Assert that we're dealing with an `HTTP/1.1` request.

4. Split the URI path on `"?"`, which gives us plain resource separate from any `GET` parameters.
5. Make a new `request` instance with the resource in place.
6. Populate that `request` instance with each split header line.
7. Set that `requests` parameters to the result of parsing our `GET` parameters.

As you might expect by now, `request` is an instance of a CLOS class:

```
(defclass request ()
  ((resource :accessor resource :initarg :resource)
   (headers :accessor headers :initarg :headers :initform nil)
   (parameters :accessor parameters :initarg :parameters :initform nil)))
```

We've now seen how our clients can send requests and have them interpreted and handled by our server. The last thing we have to implement as part of our core server interface is the capability to write responses back to the client.

Rendering Responses

Before we discuss rendering responses, we have to consider that there are two kinds of responses that we may be returning to our clients. The first is a "normal" HTTP response, complete with HTTP headers and body. We represent these kinds of responses with instances of the `response` class:

```
(defclass response ()
  ((content-type
    :accessor content-type :initform "text/html" :initarg :content-type)
   (charset
    :accessor charset :initform "utf-8")
   (response-code
    :accessor response-code :initform "200 OK" :initarg :response-code)
   (keep-alive?
    :accessor keep-alive? :initform nil :initarg :keep-alive?)
   (body
    :accessor body :initform nil :initarg :body)))
```

The second is an SSE message[15], which we will use to send an incremental update to our clients.

```
(defclass sse ()
  ((id :reader id :initarg :id :initform nil)
   (event :reader event :initarg :event :initform nil)
   (retry :reader retry :initarg :retry :initform nil)
   (data :reader data :initarg :data)))
```

We'll send an HTTP response whenever we receive a full HTTP request; however, how do we know when and where to send SSE messages without an originating client request?

A simple solution is to register *channels*[16], to which we'll subscribe `sockets` as necessary.

[15] `http://www.w3.org/TR/eventsource/`

[16] We're incidentally introducing some new syntax here. This is our way of declaring a mutable variable. It has the form `(defparameter <name> <value> <optional docstring>)`.

```
(defparameter *channels* (make-hash-table))

(defmethod subscribe! ((channel symbol) (sock usocket))
  (push sock (gethash channel *channels*))
  nil)
```

We can then `publish!` notifications to said channels as soon as they become available.

```
(defmethod publish! ((channel symbol) (message string))
  (awhen (gethash channel *channels*)
     (setf (gethash channel *channels*)
           (loop with msg = (make-instance 'sse :data message)
          for sock in it
          when (ignore-errors
             (write! msg sock)
             (force-output (socket-stream sock))
             sock)
          collect it))))
```

In `publish!`, we call `write!` to actually write an `sse` to a socket. We'll also need a specialization of `write!` on `responses` to write full HTTP responses as well. Let's handle the HTTP case first.

```
(defmethod write! ((res response) (socket usocket))
  (handler-case
      (with-timeout (.2)
    (let ((stream (flex-stream socket)))
      (flet ((write-ln (&rest sequences)
           (mapc (lambda (seq) (write-sequence seq stream)) sequences)
           (crlf stream)))
        (write-ln "HTTP/1.1 " (response-code res))
        (write-ln
         "Content-Type: " (content-type res) "; charset=" (charset res))
        (write-ln "Cache-Control: no-cache, no-store, must-revalidate")
        (when (keep-alive? res)
          (write-ln "Connection: keep-alive")
          (write-ln "Expires: Thu, 01 Jan 1970 00:00:01 GMT"))
        (awhen (body res)
          (write-ln "Content-Length: " (write-to-string (length it)))
          (crlf stream)
          (write-ln it))
        (values))))
    (trivial-timeout:timeout-error ()
      (values))))
```

This version of `write!` takes a `response` and a `usocket` named `sock`, and writes content to a stream provided by `sock`. We locally define the function `write-ln` which takes some number of sequences, and writes them out to the stream followed by a `crlf`. This is for readability; we could instead have called `write-sequence/crlf` directly.

Note that we're doing the "Must not block" thing again. While writes are likely to be buffered and are at lower risk of blocking than reads, we still don't want our server to grind to a halt if something

goes wrong here. If the write takes more than 0.2 seconds[17], we just move on (throwing out the current socket) rather than waiting any longer.

Writing an SSE out is conceptually similar to writing out a response:

```
(defmethod write! ((res sse) (socket usocket))
  (let ((stream (flex-stream socket)))
    (handler-case
    (with-timeout (.2)
      (format
       stream "~@[id: ~a~%~]~@[event: ~a~%~]~@[retry: ~a~%~]data: ~a~%~%"
       (id res) (event res) (retry res) (data res)))
      (trivial-timeout:timeout-error ()
        (values)))))
```

This is simpler than working with full HTTP responses since the SSE message standard doesn't specify CRLF line-endings, so we can get away with a single format call. The ~@[...~] blocks are *conditional directives*, which allow us to gracefully handle nil slots. For example, if (id res) is non-nil, we'll output id: <the id here>, otherwise we will ignore the directive entirely. The payload of our incremental update data is the only required slot of sse, so we can include it without worrying about it being nil. And again, we're not waiting around for *too* long. After 0.2 seconds, we'll time out and move on to the next thing if the write hasn't completed by then.

Error Responses

Our treatment of the request/response cycle so far hasn't covered what happens when something goes wrong. Specifically, we used the error! function in handle-request and process-ready without describing what it does.

```
(define-condition http-assertion-error (error)
  ((assertion :initarg :assertion :initform nil :reader assertion))
  (:report (lambda (condition stream)
         (format stream "Failed assertions '~s'"
             (assertion condition)))))
```

define-condition creates new error classes in Common Lisp. In this case, we are defining an HTTP assertion error, and stating that it will specifically need to know the actual assertion it's acting on, and a way to output itself to a stream. In other languages, you'd call this a method. Here, it's a function that happens to be the slot value of a class.

How do we represent errors to the client? Let's define the 4xx and 5xx-class HTTP errors that we'll be using often:

```
(defparameter +404+
  (make-instance
   'response :response-code "404 Not Found"
   :content-type "text/plain"
   :body "Resource not found..."))
```

[17] with-timeout has different implementations on different Lisps. In some environments, it may create another thread or process to monitor the one that invoked it. While we'd only be creating at most one of these at a time, it is a relatively heavyweight operation to be performing per-write. We might want to consider an alternative approach in those environments.

```
(defparameter +400+
  (make-instance
   'response :response-code "400 Bad Request"
   :content-type "text/plain"
   :body "Malformed, or slow HTTP request..."))

(defparameter +413+
  (make-instance
   'response :response-code "413 Request Entity Too Large"
   :content-type "text/plain"
   :body "Your request is too long..."))

(defparameter +500+
  (make-instance
   'response :response-code "500 Internal Server Error"
   :content-type "text/plain"
   :body "Something went wrong on our end..."))
```

Now we can see what `error!` does:

```
(defmethod error! ((err response) (sock usocket) &optional instance)
  (declare (ignorable instance))
  (ignore-errors
    (write! err sock)
    (socket-close sock)))
```

It takes an error response and a socket, writes the response to the socket and closes it (ignoring errors, in case the other end has already disconnected). The `instance` argument here is for logging/debugging purposes.

And with that, we have an event-driven web server that can respond to HTTP requests or send SSE messages, complete with error handling!

8.3 Extending the Server Into a Web Framework

We have now built a reasonably functional web server that will move requests, responses, and messages to and from clients. The actual work of any web application hosted by this server is done by delegating to handler functions, which were introduced in Section 8.2 but left underspecified.

The interface between our server and the hosted application is an important one, because it dictates how easily application programmers can work with our infrastructure. Ideally, our handler interface would map parameters from a request to a function that does the real work:

```
(define-handler (source :is-stream? nil) (room)
  (subscribe! (intern room :keyword) sock))

(define-handler (send-message) (room name message)
  (publish! (intern room :keyword)
        (encode-json-to-string
         `((:name . ,name) (:message . ,message)))))

(define-handler (index) ()
```

```
  (with-html-output-to-string (s nil :prologue t :indent t)
    (:html
     (:head (:script
         :type "text/javascript"
         :src "/static/js/interface.js"))
     (:body (:div :id "messages")
        (:textarea :id "input")
        (:button :id "send" "Send")))))
```

One of the concerns I had in mind when writing House was that, like any application open to the greater internet, it would be processing requests from untrusted clients. It would be nice to be able to say specifically what *type* of data each request should contain by providing a small *schema* that describes the data. Our previous list of handlers would then look like this:

```
(defun len-between (min thing max)
  (>= max (length thing) min))

(define-handler (source :is-stream? nil)
    ((room :string (len-between 0 room 16)))
  (subscribe! (intern room :keyword) sock))

(define-handler (send-message)
    ((room :string (len-between 0 room 16))
     (name :string (len-between 1 name 64))
     (message :string (len-between 5 message 256)))
  (publish! (intern room :keyword)
        (encode-json-to-string
         `((:name . ,name) (:message . ,message)))))

(define-handler (index) ()
  (with-html-output-to-string (s nil :prologue t :indent t)
    (:html
     (:head (:script
         :type "text/javascript"
         :src "/static/js/interface.js"))
     (:body (:div :id "messages")
        (:textarea :id "input")
        (:button :id "send" "Send")))))
```

While we are still working with Lisp code, this interface is starting to look almost like a *declarative language*, in which we state *what* we want our handlers to validate without thinking too much about *how* they are going to do it. What we are doing is building a *domain-specific language* (DSL) for handler functions; that is, we are creating a specific convention and syntax that allows us to concisely express exactly what we want our handlers to validate. This approach of building a small language to solve the problem at hand is frequently used by Lisp programmers, and it is a useful technique that can be applied in other programming languages.

A DSL for Handlers

Now that we have a loose specification for how we want our handler DSL to look, how do we implement it? That is, what specifically do we expect to happen when we call `define-handler`?

Let's consider the definition for `send-message` from above:

```
(define-handler (send-message)
    ((room :string (len-between 0 room 16))
     (name :string (len-between 1 name 64))
     (message :string (len-between 5 message 256)))
  (publish! (intern room :keyword)
        (encode-json-to-string
         `((:name . ,name) (:message . ,message)))))
```

What we would like `define-handler` to do here is:

1. Bind the action `(publish! ...)` to the URI `/send-message` in the handlers table.
2. When a request to this URI is made:
    - Ensure that the HTTP parameters `room`, `name` and `message` were included.
    - Validate that `room` is a string no longer than 16 characters, `name` is a string of between 1 and 64 characters (inclusive) and that `message` is a string of between 5 and 256 characters (also inclusive).
3. After the response has been returned, close the channel.

While we could write Lisp functions to do all of these things, and then manually assemble the pieces ourselves, a more common approach is to use a Lisp facility called `macros` to *generate* the Lisp code for us. This allows us to concisely express what we want our DSL to do, without having to maintain a lot of code to do it. You can think of a macro as an "executable template" that will be expanded into Lisp code at runtime.

Here's our `define-handler` macro[18]:

```
(defmacro define-handler
    ((name &key (is-stream? t) (content-type "text/html")) (&rest args)
     &body body)
  (if is-stream?
      `(bind-handler
    ,name (make-closing-handler
           (:content-type ,content-type)
           ,args ,@body))
      `(bind-handler
    ,name (make-stream-handler ,args ,@body))))
```

It delegates to three other macros (`bind-handler`, `make-closing-handler`, `make-stream-handler`) that we will define later. `make-closing-handler` will create a handler for a full HTTP request/response cycle; `make-stream-handler` will instead handle an SSE message. The predicate `is-stream?` distinguishes between these cases for us. The backtick and comma are macro-specific operators that we can use to "cut holes" in our code that will be filled out by values specified in our Lisp code when we actually use `define-handler`.

Notice how closely our macro conforms to our specification of what we wanted `define-handler` to do: If we were to write a series of Lisp functions to do all of these things, the intent of the code would be much more difficult to discern by inspection.

[18] I should note, the below code-block is VERY unconventional indentation for Common Lisp. Arglists are typically not broken up over multiple lines, and are usually kept on the same line as the macro/function name. I had to do it to stick to the line-width guidelines for this book, but would otherwise prefer to have longer lines that break naturally at places dictated by the content of the code.

Expanding a Handler

Let's step through the expansion for the `send-message` handler so that we better understand what is actually going on when Lisp "expands" our macro for us. We'll use the macro expansion feature from the SLIME[19] Emacs mode to do this. Calling `macro-expander` on `define-handler` will expand our macro by one "level", leaving our helper macros in their still-condensed form:

```
(BIND-HANDLER
 SEND-MESSAGE
 (MAKE-CLOSING-HANDLER
  (:CONTENT-TYPE "text/html")
  ((ROOM :STRING (LEN-BETWEEN 0 ROOM 16))
   (NAME :STRING (LEN-BETWEEN 1 NAME 64))
   (MESSAGE :STRING (LEN-BETWEEN 5 MESSAGE 256)))
  (PUBLISH! (INTERN ROOM :KEYWORD)
        (ENCODE-JSON-TO-STRING
         `((:NAME ,@NAME) (:MESSAGE ,@MESSAGE))))))
```

Our macro has already saved us a bit of typing by substituting our `send-message` specific code into our handler template. `bind-handler` is another macro which maps a URI to a handler function on our handlers table; since it's now at the root of our expansion, let's see how it is defined before expanding this further.

```
(defmacro bind-handler (name handler)
  (assert (symbolp name) nil "`name` must be a symbol")
  (let ((uri (if (eq name 'root) "/" (format nil "/~(~a~)" name))))
    `(progn
       (when (gethash ,uri *handlers*)
     (warn ,(format nil "Redefining handler '~a'" uri)))
       (setf (gethash ,uri *handlers*) ,handler))))
```

The binding happens in the last line: `(setf (gethash ,uri *handlers*) ,handler)`, which is what hash-table assignments look like in Common Lisp (modulo the commas, which are part of our macro). Note that the `assert` is outside of the quoted area, which means that it'll be run as soon as the macro is *called* rather than when its result is evaluated.

When we further expand our expansion of the `send-message` `define-handler` above, we get:

```
(PROGN
  (WHEN (GETHASH "/send-message" *HANDLERS*)
    (WARN "Redefining handler '/send-message'"))
  (SETF (GETHASH "/send-message" *HANDLERS*)
    (MAKE-CLOSING-HANDLER
     (:CONTENT-TYPE "text/html")
     ((ROOM :STRING (LEN-BETWEEN 0 ROOM 16))
      (NAME :STRING (LEN-BETWEEN 1 NAME 64))
      (MESSAGE :STRING (LEN-BETWEEN 5 MESSAGE 256)))
     (PUBLISH! (INTERN ROOM :KEYWORD)
           (ENCODE-JSON-TO-STRING
            `((:NAME ,@NAME) (:MESSAGE ,@MESSAGE)))))))
```

[19] `https://common-lisp.net/project/slime/`

This is starting to look more like a custom implementation of what we would have written to marshal a request from a URI to a handler function, had we written it all ourselves. But we didn't have to!

We still have `make-closing-handler` left to go in our expansion. Here is its definition:

```
(defmacro make-closing-handler
    ((&key (content-type "text/html")) (&rest args) &body body)
  `(lambda (sock parameters)
     (declare (ignorable parameters))
     ,(arguments
       args
       `(let ((res (make-instance
            'response
            :content-type ,content-type
            :body (progn ,@body))))
      (write! res sock)
      (socket-close sock)))))
```

So making a closing-handler involves making a `lambda`, which is just what you call anonymous functions in Common Lisp. We also set up an interior scope that makes a `response` out of the `body` argument we're passing in, performs a `write!` to the requesting socket, then closes it. The remaining question is, what is `arguments`?

```
(defun arguments (args body)
  (loop with res = body
     for arg in args
     do (match arg
     ((guard arg-sym (symbolp arg-sym))
      (setf res `(let ((,arg-sym ,(arg-exp arg-sym))) ,res)))
     ((list* arg-sym type restrictions)
      (setf res
        (let ((sym (or (type-expression
                 (arg-exp arg-sym)
                 type restrictions)
                    (arg-exp arg-sym))))
          `(let ((,arg-sym ,sym))
             ,@(awhen (type-assertion arg-sym type restrictions)
             `((assert-http ,it)))
             ,res)))))
     finally (return res)))
```

Welcome to the hard part. `arguments` turns the validators we registered with our handler into a tree of parse attempts and assertions. `type-expression`, `arg-exp`, and `type-assertion` are used to implement and enforce a "type system" for the kinds of data we're expecting in our responses; we'll discuss them in Section 8.3. Using this together with `make-closing-handler` would implement the validation rules we wrote here:

```
(define-handler (send-message)
    ((room :string (>= 16 (length room)))
     (name :string (>= 64 (length name) 1))
     (message :string (>= 256 (length message) 5)))
```

```
  (publish! (intern room :keyword)
        (encode-json-to-string
         `((:name . ,name) (:message . ,message)))))
```

…as an "unrolled" sequence of checks needed to validate the request:

```
(LAMBDA (SOCK #:COOKIE?1111 SESSION PARAMETERS)
  (DECLARE (IGNORABLE SESSION PARAMETERS))
  (LET ((ROOM (AIF (CDR (ASSOC :ROOM PARAMETERS))
           (URI-DECODE IT)
           (ERROR (MAKE-INSTANCE
               'HTTP-ASSERTION-ERROR
               :ASSERTION 'ROOM)))))
    (ASSERT-HTTP (>= 16 (LENGTH ROOM)))
    (LET ((NAME (AIF (CDR (ASSOC :NAME PARAMETERS))
             (URI-DECODE IT)
             (ERROR (MAKE-INSTANCE
                 'HTTP-ASSERTION-ERROR
                 :ASSERTION 'NAME)))))
      (ASSERT-HTTP (>= 64 (LENGTH NAME) 1))
      (LET ((MESSAGE (AIF (CDR (ASSOC :MESSAGE PARAMETERS))
               (URI-DECODE IT)
               (ERROR (MAKE-INSTANCE
                   'HTTP-ASSERTION-ERROR
                   :ASSERTION 'MESSAGE)))))
    (ASSERT-HTTP (>= 256 (LENGTH MESSAGE) 5))
    (LET ((RES (MAKE-INSTANCE
            'RESPONSE :CONTENT-TYPE "text/html"
            :COOKIE (UNLESS #:COOKIE?1111
                  (TOKEN SESSION))
            :BODY (PROGN
                (PUBLISH!
                 (INTERN ROOM :KEYWORD)
                 (ENCODE-JSON-TO-STRING
                  `((:NAME ,@NAME)
                (:MESSAGE ,@MESSAGE))))))))
      (WRITE! RES SOCK)
      (SOCKET-CLOSE SOCK))))))
```

This gets us the validation we need for full HTTP request/response cycles. What about our SSEs? `make-stream-handler` does the same basic thing as `make-closing-handler`, except that it writes an `SSE` rather than a `RESPONSE`, and it calls `force-output` instead of `socket-close` because we want to flush data over the connection without closing it:

```
(defmacro make-stream-handler ((&rest args) &body body)
  `(lambda (sock parameters)
     (declare (ignorable parameters))
     ,(arguments
       args
       `(let ((res (progn ,@body)))
      (write! (make-instance
           'response
```

```
           :keep-alive? t
           :content-type "text/event-stream")
          sock)
      (write!
       (make-instance 'sse :data (or res "Listening..."))
       sock)
      (force-output
       (socket-stream sock))))))

(defmacro assert-http (assertion)
  `(unless ,assertion
     (error (make-instance
         'http-assertion-error
         :assertion ',assertion))))
```

`assert-http` is a macro that creates the boilerplate code we need in error cases. It expands into a check of the given assertion, throws an `http-assertion-error` if it fails, and packs the original assertion along in that event.

```
(defmacro assert-http (assertion)
  `(unless ,assertion
     (error (make-instance
         'http-assertion-error
         :assertion ',assertion))))
```

HTTP “Types”HTTP Types

In the previous section, we briefly touched on three expressions that we’re using to implement our HTTP type validation system: `arg-exp`, `type-expression` and `type-assertion`. Once you understand those, there will be no magic left in our framework. We’ll start with the easy one first.

arg-exp

`arg-exp` takes a symbol and creates an `aif` expression that checks for the presence of a parameter.

```
(defun arg-exp (arg-sym)
  `(aif (cdr (assoc ,(->keyword arg-sym) parameters))
    (uri-decode it)
    (error (make-instance
        'http-assertion-error
        :assertion ',arg-sym))))
```

Evaluating `arg-exp` on a symbol looks like:

```
HOUSE> (arg-exp 'room)
(AIF (CDR (ASSOC :ROOM PARAMETERS))
     (URI-DECODE IT)
     (ERROR (MAKE-INSTANCE
         'HTTP-ASSERTION-ERROR
         :ASSERTION 'ROOM)))
HOUSE>
```

We've been using forms like `aif` and `awhen` without understanding how they work, so let's take some time to explore them now.

Recall that Lisp code is itself represented as a tree. That's what the parentheses are for; they show us how leaves and branches fit together. If we step back to what we were doing in the previous section, `make-closing-handler` calls a function called `arguments` to generate part of the Lisp tree it's constructing, which in turn calls some tree-manipulating helper functions, including `arg-exp`, to generate its return value.

That is, we've built a small system that takes a Lisp expression as input, and produces a different Lisp expression as output. Possibly the simplest way of conceptualizing this is as a simple Common–Lisp-to-Common–Lisp compiler that is specialized to the problem at hand.

A widely used classification of such compilers is as *anaphoric macros*. This term comes from the linguistic concept of an *anaphor*, which is the use of one word as a substitute for a group of words that preceded it. `aif` and `awhen` are anaphoric macros, and they're the only ones that I tend to often use. There are many more availabile in the `anaphora` package[20].

As far as I know, anaphoric macros were first defined by Paul Graham in an OnLisp chapter[21]. The use case he gives is a situation where you want to do some sort of expensive or semi-expensive check, then do something conditionally on the result. In the above context, we're using `aif` to do a check the result of an `alist` traversal.

```
(aif (cdr (assoc :room parameters))
     (uri-decode it)
     (error (make-instance
         'http-assertion-error
         :assertion 'room)))
```

This takes the `cdr` of looking up the symbol `:room` in the association list `parameters`. If that returns a non-nil value, `uri-decode` it, otherwise throw an error of the type `http-assertion-error`.

In other words, the above is equivalent to:

```
(let ((it (cdr (assoc :room parameters))))
  (if it
      (uri-decode it)
      (error (make-instance
          'http-assertion-error
          :assertion 'room))))
```

Strongly-typed functional languages like Haskell often use a `Maybe` type in this situation. In Common Lisp, we capture the symbol `it` in the expansion as the name for the result of the check.

Understanding this, we should be able to see that `arg-exp` is generating a specific, repetitive, piece of the code tree that we eventually want to evaluate. In this case, the piece that checks for the presence of the given parameter among the handlers' `parameters`. Now, let's move onto…

type-expression

```
(defgeneric type-expression (parameter type)
  (:documentation
   "A type-expression will tell the server
```

[20] `http://www.cliki.net/Anaphora`

[21] `http://dunsmor.com/lisp/onlisp/onlisp_18.html`

```
how to convert a parameter from a string to
a particular, necessary type."))
...
(defmethod type-expression (parameter type) nil)
```

This is a generic function that generates new tree structures (coincidentally Lisp code), rather than just a function. The only thing the above tells you is that by default, a `type-expression` is `NIL`. Which is to say, we don't have one. If we encounter a `NIL`, we use the raw output of `arg-exp`, but that doesn't tell us much about the most common case. To see that, let's take a look at a built-in (to `:house`) `define-http-type` expression.

```
(define-http-type (:integer)
    :type-expression `(parse-integer ,parameter :junk-allowed t)
    :type-assertion `(numberp ,parameter))
```

An `:integer` is something we're making from a `parameter` by using `parse-integer`. The `junk-allowed` parameter tells `parse-integer` that we're not confident the data we're giving it is actually parseable, so we need to make sure that the returned result is an integer. If it isn't, we get this behaviour:

```
HOUSE> (type-expression 'blah :integer)
(PARSE-INTEGER BLAH :JUNK-ALLOWED T)
HOUSE>
```

`define-http-handler`[22] is one of the exported symbols for our framework. This lets our application programmers define their own types to simplify parsing above the handful of "builtins" that we give them (`:string`, `:integer`, `:keyword`, `:json`, `:list-of-keyword` and `:list-of-integer`).

```
(defmacro define-http-type ((type) &key type-expression type-assertion)
  (with-gensyms (tp)
    `(let ((,tp ,type))
       ,@(when type-expression
      `((defmethod type-expression (parameter (type (eql ,tp)))
          ,type-expression)))
       ,@(when type-assertion
      `((defmethod type-assertion (parameter (type (eql ,tp)))
          ,type-assertion))))))
```

It works by creating `type-expression` and `type-assertion` method definitions for the type being defined. We could let users of our framework do this manually without much trouble; however, adding this extra level of indirection gives us, the framework programmers, the freedom to change *how* types are implemented without forcing our users to re-write their specifications. This isn't just an academic consideration; I've personally made radical changes to this part of the system when first building it, and was pleased to find that I had to make very few edits to the applications that depended on it.

Let's take a look at the expansion of that integer definition to see how it works in detail:

[22] This macro is difficult to read because it tries hard to make its output human-readable, by expanding `NIL`s away using `,@` where possible.

```
(LET ((#:TP1288 :INTEGER))
  (DEFMETHOD TYPE-EXPRESSION (PARAMETER (TYPE (EQL #:TP1288)))
    `(PARSE-INTEGER ,PARAMETER :JUNK-ALLOWED T))
  (DEFMETHOD TYPE-ASSERTION (PARAMETER (TYPE (EQL #:TP1288)))
    `(NUMBERP ,PARAMETER)))
```

As we said, it doesn't reduce code size by much, but it does prevent us from needing to care what the specific parameters of those methods are, or even that they're methods at all.

type-assertion

Now that we can define types, let's look at how we use `type-assertion` to validate that a parse satisfies our requirements. It, too, takes the form of a complementary `defgeneric`/`defmethod` pair just like `type-expression`:

```
(defgeneric type-assertion (parameter type)
  (:documentation
   "A lookup assertion is run on a parameter
immediately after conversion. Use it to restrict
 the space of a particular parameter."))
...
(defmethod type-assertion (parameter type) nil)
```

Here's what this one outputs:

```
HOUSE> (type-assertion 'blah :integer)
(NUMBERP BLAH)
HOUSE>
```

There are cases where `type-assertion` won't need to do anything. For example, since HTTP parameters are given to us as strings, our `:string` type assertion has nothing to validate:

```
HOUSE> (type-assertion 'blah :string)
NIL
HOUSE>
```

All Together Now

We did it! We built a web framework on top of an event-driven webserver implementation. Our framework (and handler DSL) defines new applications by:

- Mapping URLs to handlers;
- Defining handlers to enforce the type safety and validation rules on requests;
- Optionally specifying new types for handlers as required.

Now we can describe our application like this:

```
(defun len-between (min thing max)
  (>= max (length thing) min))

(define-handler (source :is-stream? nil)
    ((room :string (len-between 0 room 16)))
```

```
  (subscribe! (intern room :keyword) sock))

(define-handler (send-message)
    ((room :string (len-between 0 room 16))
     (name :string (len-between 1 name 64))
     (message :string (len-between 5 message 256)))
  (publish! (intern room :keyword)
        (encode-json-to-string
         `((:name . ,name) (:message . ,message)))))

(define-handler (index) ()
  (with-html-output-to-string (s nil :prologue t :indent t)
    (:html
     (:head (:script
         :type "text/javascript"
         :src "/static/js/interface.js"))
     (:body (:div :id "messages")
        (:textarea :id "input")
        (:button :id "send" "Send")))))

(start 4242)
```

Once we write `interface.js` to provide the client-side interactivity, this will start an HTTP chat server on port `4242` and listen for incoming connections.

Making Your Own Image Filters

Cate Huston

11.1 A Brilliant Idea (That Wasn't All That Brilliant)

When I was traveling in China I often saw series of four paintings showing the same place in different seasons. Color—the cool whites of winter, pale hues of spring, lush greens of summer, and reds and yellows of fall—is what visually differentiates the seasons. Around 2011, I had what I thought was a brilliant idea: I wanted to be able to visualize a photo series as a series of colors. I thought it would show travel, and progression through the seasons.

But I didn't know how to calculate the dominant color from an image. I thought about scaling the image down to a 1x1 square and seeing what was left, but that seemed like cheating. I knew how I wanted to display the images, though: in a layout called the Sunflower layout[1]. It's the most efficient way to lay out circles.

I left this project for years, distracted by work, life, travel, talks. Eventually I returned to it, figured out how to calculate the dominant color, and finished my visualization[2]. That is when I discovered that this idea wasn't, in fact, brilliant. The progression wasn't as clear as I hoped, the dominant color extracted wasn't generally the most appealing shade, the creation took a long time (a couple of seconds per image), and it took hundreds of images to make something cool (Figure 11.1).

You might think this would be discouraging, but by the time I got to this point I had learned many things that hadn't come my way before — about color spaces and pixel manipulation — and I had started making those cool partially colored images, the kind you find on postcards of London with a red bus or phone booth and everything else in grayscale.

I used a framework called Processing[3] because I was familiar with it from developing programming curricula, and because I knew it made it easy to create visual applications. It's a tool originally designed for artists, so it abstracts away much of the boilerplate. It allowed me to play and experiment.

University, and later work, filled up my time with other people's ideas and priorities. Part of finishing this project was learning how to carve out time to make progress on my own ideas; I required about four hours of good mental time a week. A tool that allowed me to move faster was therefore really helpful, even necessary—although it came with its own set of problems, especially around writing tests.

I felt that thorough tests were especially important for validating how the project was working, and for making it easier to pick up and resume a project that was often on ice for weeks, even months

[1] http://www.catehuston.com/applets/Sunflower/index.html
[2] http://www.catehuston.com/blog/2013/09/02/visualising-a-photo-series/
[3] https://processing.org/

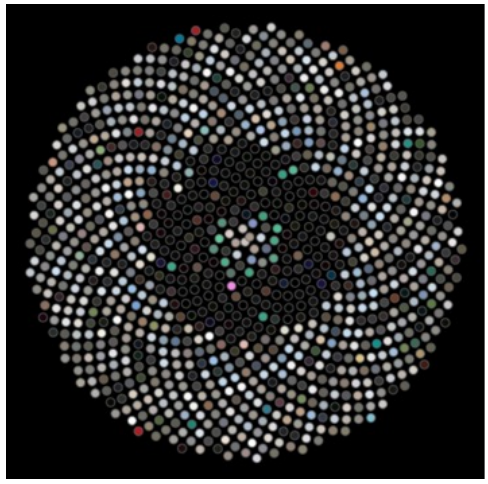

Figure 11.1: Sunflower layout

at a time. Tests (and blogposts!) formed the documentation for this project. I could leave failing tests to document what should happen that I hadn't figured out yet, and make changes with confidence that if I changed something that I had forgotten was critical, the tests would remind me.

This chapter will cover some details about Processing and talk you through color spaces, decomposing an image into pixels and manipulating them, and unit testing something that wasn't designed with testing in mind. But I hope it will also prompt you to make some progress on whatever idea you haven't made time for lately; even if your idea turns out to be as terrible as mine was, you may make something cool and learn something fascinating in the process.

11.2 The App

This chapter will show you how to create an image filter application that you can use to manipulate your digital images using filters that you create. We'll use Processing, a programming language and development environment built in Java. We'll cover setting up the application in Processing, some of the features of Processing, aspects of color representation, and how to create color filters (mimicking what was used in old-fashioned photography). We'll also create a special kind of filter that can only be done digitally: determining the dominant hue of an image and showing or hiding it, to create eerie partially colored images.

Finally, we'll add a thorough test suite, and cover how to handle some of the limitations of Processing when it comes to testability.

11.3 Background

Today we can take a photo, manipulate it, and share it with all our friends in a matter of seconds. However, a long long time ago (in digital terms), it was a process that took weeks.

In the old days, we would take the picture, then when we had used a whole roll of film, we would take it in to be developed (often at the pharmacy). We'd pick up the developed pictures some days later—and discover that there was something wrong with many of them. Hand not steady enough?

Random person or thing that we didn't notice at the time? Overexposed? Underexposed? Of course by then it was too late to remedy the problem.

The process that turned the film into pictures was one that most people didn't understand. Light was a problem, so you had to be careful with the film. There was a process, involving darkened rooms and chemicals, that they sometimes showed in films or on TV.

But probably even fewer people understand how we get from the point-and-click on our smartphone camera to an image on Instagram. There are actually many similarities.

Photographs, the Old Way

Photographs are created by the effect of light on a light-sensitive surface. Photographic film is covered in silver halide crystals. (Extra layers are used to create color photographs — for simplicity let's just stick to black-and-white photography here.)

When talking an old-fashioned photograph — with film — the light hits the film according to what you're pointing at, and the crystals at those points are changed in varying degrees, according to the amount of light. Then, the development process[4] converts the silver salts to metallic silver, creating the negative. The negative has the light and dark areas of the image inverted. Once the negatives have been developed, there is another series of steps to reverse the image and print it.

Photographs, the Digital Way

When taking pictures using our smartphones or digital cameras, there is no film. There is something called an *active-pixel sensor* which functions in a similar way. Where we used to have silver crystals, now we have pixels — tiny squares. (In fact, pixel is short for "picture element".) Digital images are made up of pixels, and the higher the resolution the more pixels there are. This is why low-resolution images are described as "pixelated" — you can start to see the squares. These pixels are stored in an array, with the number in each array "box" containing the color.

In Figure 11.2, we see a high-resolution picture of some blow-up animals taken at MoMA in NYC. Figure 11.3 is the same image blown up, but with just 24 x 32 pixels.

See how it's so blurry? We call that *pixelation*, which means the image is too big for the number of pixels it contains and the squares become visible. Here we can use it to get a better sense of an image being made up of squares of color.

What do these pixels look like? If we print out the colors of some of the pixels in the middle (10,10 to 10,14) using the handy `Integer.toHexString` in Java, we get hex colors:

```
FFE8B1
FFFAC4
FFFCC3
FFFCC2
FFF5B7
```

Hex colors are six characters long. The first two are the red value, the second two the green value, and the third two the blue value. Sometimes there are an extra two characters which are the alpha value. In this case `FFFAC4` means:

[4] `http://photography.tutsplus.com/tutorials/step-by-step-guide-to-developing-black-and-white-t-max-film--photo-2580`

Figure 11.2: Blow-up animals at MoMA NY

- red = FF (hex) = 255 (base 10)
- green = FA (hex) = 250 (base 10)
- blue = C4 (hex) = 196 (base 10)

11.4 Running the App

In Figure 11.4, we have a picture of our app running. It's very much developer-designed, I know, but we only have 500 lines of Java to work with so something had to suffer! You can see the list of commands on the right. Some things we can do:

- Adjust the RGB filters.
- Adjust the "hue tolerance".
- Set the dominant hue filters, to either show or hide the dominant hue.
- Apply our current setting (it is infeasible to run this every key press).
- Reset the image.
- Save the image we have made.

Processing makes it simple to create a little application and do image manipulation; it has a very visual focus. We'll work with the Java-based version, although Processing has now been ported to other languages.

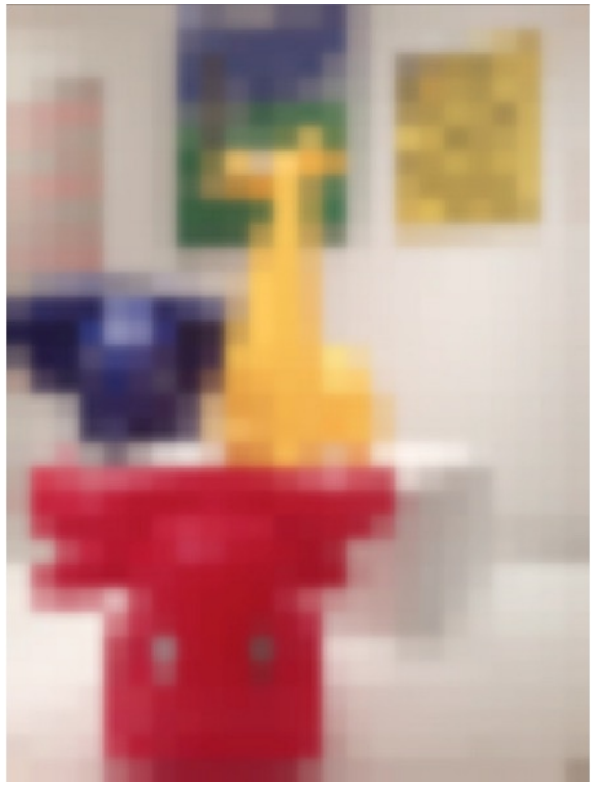

Figure 11.3: Blow-up animals, blown up

For this tutorial, I use Processing in Eclipse by adding `core.jar` to my build path. If you want, you can use the Processing IDE, which removes the need for a lot of boilerplate Java code. If you later want to port it over to Processing.js and upload it online, you need to replace the file chooser with something else.

There are detailed instructions with screenshots in the project's repository[5]. If you are familiar with Eclipse and Java already you may not need them.

11.5 Processing Basics

Size and Color

We don't want our app to be a tiny grey window, so the two essential methods that we will start by overriding are `setup()`[6], and `draw()`[7]. The `setup()` method is only called when the app starts, and is where we do things like set the size of the app window. The `draw()` method is called for every animation, or after some action can be triggered by calling `redraw()`. (As covered in the Processing Documentation, `draw()` should not be called explicitly.)

[5] `https://github.com/aosabook/500lines/blob/master/image-filters/SETUP.MD`
[6] `http://processing.org/reference/setup_.html`
[7] `http://processing.org/reference/draw_.html`

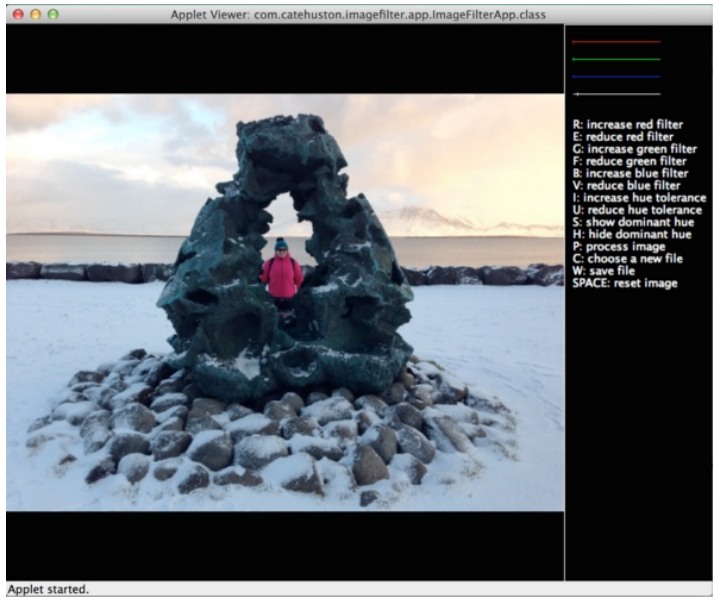

Figure 11.4: The App

Processing is designed to work nicely to create animated sketches, but in this case we don't want animation[8], we want to respond to key presses. To prevent animation (which would be a drag on performance) we will call `noLoop()`[9] from setup. This means that `draw()` will only be called immediately after `setup()`, and whenever we call `redraw()`.

```
private static final int WIDTH = 360;
private static final int HEIGHT = 240;

public void setup() {
  noLoop();

  // Set up the view.
  size(WIDTH, HEIGHT);
  background(0);
}

public void draw() {
  background(0);
}
```

These don't really do much yet, but try running the app again, adjusting the constants in `WIDTH` and `HEIGHT`, to see different sizes.

`background(0)` specifies a black background. Try changing the number passed to `background()` and see what happens — it's the alpha value, and so if you only pass one number in, it is always greyscale. Alternatively, you can call `background(int r, int g, int b)`.

[8]If we wanted to create an animated sketch we would not call `noLoop()` (or, if we wanted to start animating later, we would call `loop()`). The frequency of the animation is determined by `frameRate()`.
[9]`http://www.processing.org/reference/noLoop_.html`

pixels[]	Array containing the color of every pixel in the image
width	Image width in pixels
height	Image height in pixels

Table 11.1: PImage fields

loadPixels	Loads the pixel data for the image into its ‘pixels[]‘ array
updatePixels	Updates the image with the data in its ‘pixels[]‘ array
resize	Changes the size of an image to a new width and height
get	Reads the color of any pixel or grabs a rectangle of pixels
set	Writes a color to any pixel or writes an image into another
save	Saves the image to a TIFF, TARGA, PNG, or JPEG file

Table 11.2: PImage methods

PImage

The PImage object[10] is the Processing object that represents an image. We're going to be using this a lot, so it's worth reading through the documentation. It has three fields (Table 11.1) as well as some methods that we will use (Table 11.2).

File Chooser

Processing handles most of the file choosing process; we just need to call `selectInput()`[11], and implement a callback (which must be public).

To people familiar with Java this might seem odd; a listener or a lambda expression might make more sense. However, as Processing was developed as a tool for artists, for the most part these things have been abstracted away by the language to keep it unintimidating. This is a choice the designers made: to prioritize simplicity and approachability over power and flexibility. If you use the stripped-down Processing editor, rather than Processing as a library in Eclipse, you don't even need to define class names.

Other language designers with different target audiences make different choices, as they should. For example, in Haskell, a purely functional language, purity of functional language paradigms is prioritised over everything else. This makes it a better tool for mathematical problems than for anything requiring IO.

```
// Called on key press.
private void chooseFile() {
  // Choose the file.
  selectInput("Select a file to process:", "fileSelected");
}

public void fileSelected(File file) {
  if (file == null) {
    println("User hit cancel.");
  } else {
    // save the image
    redraw(); // update the display
  }
}
```

[10] `http://processing.org/reference/PImage.html`
[11] `http://www.processing.org/reference/selectInput_.html`

Responding to Key Presses

Normally in Java, responding to key presses requires adding listeners and implementing anonymous functions. However, as with the file chooser, Processing handles a lot of this for us. We just need to implement `keyPressed()`[12].

```
public void keyPressed() {
  print("key pressed: " + key);
}
```

If you run the app again, every time you press a key it will output it to the console. Later, you'll want to do different things depending on what key was pressed, and to do this you just switch on the key value. (This exists in the `PApplet` superclass, and contains the last key pressed.)

11.6 Writing Tests

This app doesn't do a lot yet, but we can already see number of places where things can go wrong; for example, triggering the wrong action with key presses. As we add complexity, we add more potential problems, such as updating the image state incorrectly, or miscalculating pixel colors after applying a filter. I also just enjoy (some think weirdly) writing unit tests. Whilst some people seem to think of testing as a thing that delays checking code in, I see tests as my #1 debugging tool, and as an opportunity to deeply understand what is going on in my code.

I adore Processing, but it's designed to create visual applications, and in this area maybe unit testing isn't a huge concern. It's clear it isn't written for testability; in fact it's written in such a way that makes it untestable, as is. Part of this is because it hides complexity, and some of that hidden complexity is really useful in writing unit tests. The use of static and final methods make it much harder to use mocks (objects that record interaction and allow you to fake part of your system to verify another part is behaving correctly), which rely on the ability to subclass.

We might start a greenfield project with great intentions to do Test Driven Development (TDD) and achieve perfect test coverage, but in reality we are usually looking at a mass of code written by various and assorted people and trying to figure out what it is supposed to be doing, and how and why it is going wrong. Then maybe we don't write perfect tests, but writing tests at all will help us navigate the situation, document what is happening and move forward.

We create "seams" that allow us to break something up from its amorphous mass of tangled pieces and verify it in parts. To do this, we will sometimes create wrapper classes that can be mocked. These classes do nothing more than hold a collection of similar methods, or forward calls on to another object that cannot be mocked (due to final or static methods), and as such they are very dull to write, but key to creating seams and making the code testable.

I used JUnit for tests, as I was working in Java with Processing as a library. For mocking I used Mockito. You can download Mockito[13] and add the JAR to your buildpath in the same way you added `core.jar`. I created two helper classes that make it possible to mock and test the app (otherwise we can't test behavior involving `PImage` or `PApplet` methods).

`IFAImage` is a thin wrapper around PImage. `PixelColorHelper` is a wrapper around applet pixel color methods. These wrappers call the final, and static methods, but the caller methods are neither final nor static themselves — this allows them to be mocked. These are deliberately lightweight, and

[12] `https://www.processing.org/reference/keyPressed_.html`
[13] `https://code.google.com/p/mockito/downloads/list`

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Table 11.3: Pixel indices for a 4x4 image

we could have gone further, but this was sufficient to address the major problem of testability when using Processing — static, and final methods. The goal was to make an app, after all — not a unit testing framework for Processing!

A class called `ImageState` forms the "model" of this application, removing as much logic from the class extending `PApplet` as possible, for better testability. It also makes for a cleaner design and separation of concerns: the `App` controls the interactions and the UI, not the image manipulation.

11.7 Do-It-Yourself Filters

RGB Filters

Before we start writing more complicated pixel processing, we can start with a short exercise that will get us comfortable doing pixel manipulation. We'll create standard (red, green, blue) color filters that will allow us to create the same effect as placing a colored plate over the lens of a camera, only letting through light with enough red (or green, or blue).

By applying different RGB filters to an image we can make it almost seem like the seasons are different depending which colors are filtered out and which are emphasized. (Remember the four-seasons paintings we imagined earlier?)

How do we do it?

- Set the filter. (You can combine red, green and blue filters as in the image earlier; I haven't in these examples so that the effect is clearer.)
- For each pixel in the image, check its RGB value.
- If the red is less than the red filter, set the red to zero.
- If the green is less than the green filter, set the green to zero.
- If the blue is less than the blue filter, set the blue to zero.
- Any pixel with insufficient of all of these colors will be black.

Although our image is 2-dimensional, the pixels live in a 1-dimensional array starting top-left and moving left to right, top to bottom[14]. The array indices for a 4x4 image are shown here:

```
public void applyColorFilter(PApplet applet, IFAImage img, int minRed,
      int minGreen, int minBlue, int colorRange) {
  img.loadPixels();
  int numberOfPixels = img.getPixels().length;
  for (int i = 0; i < numberOfPixels; i++) {
    int pixel = img.getPixel(i);
    float alpha = pixelColorHelper.alpha(applet, pixel);
    float red = pixelColorHelper.red(applet, pixel);
    float green = pixelColorHelper.green(applet, pixel);
    float blue = pixelColorHelper.blue(applet, pixel);
```

[14] https://processing.org/tutorials/pixels/

```
    red = (red >= minRed) ? red : 0;
    green = (green >= minGreen) ? green : 0;
    blue = (blue >= minBlue) ? blue : 0;

    image.setPixel(i, pixelColorHelper.color(applet, red, green, blue, alpha));
  }
}
```

Color

As our first example of an image filter showed, the concept and representation of colors in a program is very important to understanding how our filters work. To prepare ourselves for working on our next filter, let's explore the concept of color a bit more.

We were using a concept in the previous section called "color space", which is way of representing color digitally. Kids mixing paints learn that colors can be made from other colors; things work slightly differently in digital (less risk of being covered in paint!) but similar. Processing makes it really easy to work with whatever color space you want, but you need to know which one to pick, so it's important to understand how they work.

RGB colors

The color space that most programmers are familiar with is RGBA: red, green, blue and alpha; it's what we were using above. In hexadecimal (base 16), the first two digits are the amount of red, the second two blue, the third two green, and the final two (if they are there) are the alpha value. The values range from 00 in base 16 (0 in base 10) through to FF (255 in base 10). The alpha represents opacity, where 0 is transparent and 100% is opaque.

HSB or HSV colors

This color space is not quite as well known as RGB. The first number represents the hue, the second number the saturation (how intense the color is), and the third number the brightness. The HSB color space can be represented by a cone: The hue is the position around the cone, saturation the distance from the centre, and brightness the height (0 brightness is black).

Extracting the Dominant Hue from an Image

Now that we're comfortable with pixel manipulation, let's do something that we could only do digitally. Digitally, we can manipulate the image in a way that isn't so uniform.

When I look through my stream of pictures I can see themes emerging. The nighttime series I took at sunset from a boat on Hong Kong harbour, the grey of North Korea, the lush greens of Bali, the icy whites and pale blues of an Icelandic winter. Can we take a picture and pull out that main color that dominates the scene?

It makes sense to use the HSB color space for this — we are interested in the hue when figuring out what the main color is. It's possible to do this using RGB values, but more difficult (we would have to compare all three values) and it would be more sensitive to darkness. We can change to the HSB color space using colorMode[15].

[15] `http://processing.org/reference/colorMode_.html`

Having settled on this color space, it's simpler than it would have been using RGB. We need to find the hue of each pixel, and figure out which is most "popular". We probably don't want to be exact — we want to group very similar hues together, and we can handle this using two strategies.

Firstly we will round the decimals that come back to whole numbers, as this makes it simple to determine which "bucket" we put each pixel in. Secondly we can change the range of the hues. If we think back to the cone representation above, we might think of hues as having 360 degrees (like a circle). Processing uses 255 by default, which is the same as is typical for RGB (255 is FF in hexadecimal). The higher the range we use, the more distinct the hues in the picture will be. Using a smaller range will allow us to group together similar hues. Using a 360 degree range, it's unlikely that we will be able to tell the difference between a hue of 224 and a hue of 225, as the difference is very small. If we make the range one-third of that, 120, both these hues become 75 after rounding.

We can change the range of hues using `colorMode`. If we call `colorMode(HSB, 120)` we have just made our hue detection a bit less than half as exact as if we used the 255 range. We also know that our hues will fall into 120 "buckets", so we can simply go through our image, get the hue for a pixel, and add one to the corresponding count in an array. This will be $O(n)$, where n is the number of pixels, as it requires action on each one.

```
for(int px in pixels) {
  int hue = Math.round(hue(px));
  hues[hue]++;
}
```

At the end we can print this hue to the screen, or display it next to the picture.

Once we've extracted the "dominant" hue, we can choose to either show or hide it in the image. We can show the dominant hue with varying tolerance (ranges around it that we will accept). Pixels that don't fall into this range can be changed to grayscale by setting the value based on the brightness. Alternatively, we can hide the dominant hue by setting the color for pixels with that hue to greyscale, and leaving other pixels as they are.

Each image requires a double pass (looking at each pixel twice), so on images with a large number of pixels it can take a noticeable amount of time.

```
public HSBColor getDominantHue(PApplet applet, IFAImage image, int hueRange) {
  image.loadPixels();
  int numberOfPixels = image.getPixels().length;
  int[] hues = new int[hueRange];
  float[] saturations = new float[hueRange];
  float[] brightnesses = new float[hueRange];

  for (int i = 0; i < numberOfPixels; i++) {
    int pixel = image.getPixel(i);
    int hue = Math.round(pixelColorHelper.hue(applet, pixel));
    float saturation = pixelColorHelper.saturation(applet, pixel);
    float brightness = pixelColorHelper.brightness(applet, pixel);
    hues[hue]++;
    saturations[hue] += saturation;
    brightnesses[hue] += brightness;
  }

  // Find the most common hue.
  int hueCount = hues[0];
```

```
  int hue = 0;
  for (int i = 1; i < hues.length; i++) {
    if (hues[i] > hueCount) {
      hueCount = hues[i];
      hue = i;
    }
  }

  // Return the color to display.
  float s = saturations[hue] / hueCount;
  float b = brightnesses[hue] / hueCount;
  return new HSBColor(hue, s, b);
}


public void processImageForHue(PApplet applet, IFAImage image, int hueRange,
    int hueTolerance, boolean showHue) {
  applet.colorMode(PApplet.HSB, (hueRange - 1));
  image.loadPixels();
  int numberOfPixels = image.getPixels().length;
  HSBColor dominantHue = getDominantHue(applet, image, hueRange);
  // Manipulate photo, grayscale any pixel that isn't close to that hue.
  float lower = dominantHue.h - hueTolerance;
  float upper = dominantHue.h + hueTolerance;
  for (int i = 0; i < numberOfPixels; i++) {
    int pixel = image.getPixel(i);
    float hue = pixelColorHelper.hue(applet, pixel);
    if (hueInRange(hue, hueRange, lower, upper) == showHue) {
      float brightness = pixelColorHelper.brightness(applet, pixel);
      image.setPixel(i, pixelColorHelper.color(applet, brightness));
    }
  }
  image.updatePixels();
}
```

Combining Filters

With the UI as it is, the user can combine the red, green, and blue filters together. If they combine the dominant hue filters with the red, green, and blue filters the results can sometimes be a little unexpected, because of changing the color spaces.

Processing has some built-in methods[16] that support the manipulation of images; for example, `invert` and `blur`.

To achieve effects like sharpening, blurring, or sepia we apply matrices. For every pixel of the image, take the sum of products where each product is the color value of the current pixel or a neighbor of it, with the corresponding value of the filter matrix[17]. There are some special matrices of specific values that sharpen images.

[16] `https://www.processing.org/reference/filter_.html`
[17] `http://lodev.org/cgtutor/filtering.html`

11.8 Architecture

There are three main components to the app (Figure 11.5).

The App

The app consists of one file: `ImageFilterApp.java`. This extends `PApplet` (the Processing app superclass) and handles layout, user interaction, etc. This class is the hardest to test, so we want to keep it as small as possible.

Model

Model consists of three files: `HSBColor.java` is a simple container for HSB colors (consisting of hue, saturation, and brightness). `IFAImage` is a wrapper around `PImage` for testability. (`PImage` contains a number of final methods which cannot be mocked.) Finally, `ImageState.java` is the object which describes the state of the image — what level of filters should be applied, and which filters — and handles loading the image. (Note: The image needs to be reloaded whenever color filters are adjusted down, and whenever the dominant hue is recalculated. For clarity, we just reload each time the image is processed.)

Color

Color consists of two files: `ColorHelper.java` is where all the image processing and filtering takes place, and `PixelColorHelper.java` abstracts out final `PApplet` methods for pixel colors for testability.

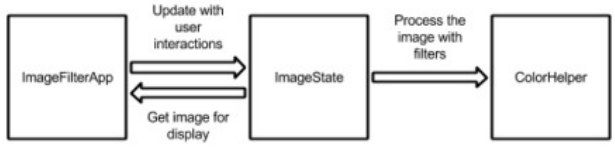

Figure 11.5: Architecture diagram

Wrapper Classes and Tests

Briefly mentioned above, there are two wrapper classes (`IFAImage` and `PixelColorHelper`) that wrap library methods for testability. This is because, in Java, the keyword "final" indicates a method that cannot be overridden or hidden by subclasses, which means they cannot be mocked.

`PixelColorHelper` wraps methods on the applet. This means we need to pass the applet in to each method call. (Alternatively, we could make it a field and set it on initialization.)

```
package com.catehuston.imagefilter.color;

import processing.core.PApplet;

public class PixelColorHelper {
```

```
  public float alpha(PApplet applet, int pixel) {
    return applet.alpha(pixel);
  }

  public float blue(PApplet applet, int pixel) {
    return applet.blue(pixel);
  }

  public float brightness(PApplet applet, int pixel) {
    return applet.brightness(pixel);
  }

  public int color(PApplet applet, float greyscale) {
    return applet.color(greyscale);
  }

  public int color(PApplet applet, float red, float green, float blue,
           float alpha) {
    return applet.color(red, green, blue, alpha);
  }

  public float green(PApplet applet, int pixel) {
    return applet.green(pixel);
  }

  public float hue(PApplet applet, int pixel) {
    return applet.hue(pixel);
  }

  public float red(PApplet applet, int pixel) {
    return applet.red(pixel);
  }

  public float saturation(PApplet applet, int pixel) {
    return applet.saturation(pixel);
  }
}
```

`IFAImage` is a wrapper around `PImage`, so in our app we don't initialize a `PImage`, but rather an `IFAImage` — although we do have to expose the `PImage` so that it can be rendered.

```
package com.catehuston.imagefilter.model;

import processing.core.PApplet;
import processing.core.PImage;

public class IFAImage {

  private PImage image;

  public IFAImage() {
    image = null;
```

```
  }

  public PImage image() {
    return image;
  }

  public void update(PApplet applet, String filepath) {
    image = null;
    image = applet.loadImage(filepath);
  }

  // Wrapped methods from PImage.
  public int getHeight() {
    return image.height;
  }

  public int getPixel(int px) {
    return image.pixels[px];
  }

  public int[] getPixels() {
    return image.pixels;
  }

  public int getWidth() {
    return image.width;
  }

  public void loadPixels() {
    image.loadPixels();
  }

  public void resize(int width, int height) {
    image.resize(width, height);
  }

  public void save(String filepath) {
    image.save(filepath);
  }

  public void setPixel(int px, int color) {
    image.pixels[px] = color;
  }

  public void updatePixels() {
    image.updatePixels();
  }
}
```

Finally, we have our simple container class, `HSBColor`. Note that it is immutable (once created, it cannot be changed). Immutable objects are better for thread safety (something we have no need of

here!) but are also easier to understand and reason about. In general, I tend to make simple model classes immutable unless I find a good reason for them not to be.

Some of you may know that there are already classes representing color in Processing[18] and in Java itself[19]. Without going too much into the details of these, both of them are more focused on RGB color, and the Java class in particular adds way more complexity than we need. We would probably be okay if we did want to use Java's `awt.Color`; however awt GUI components cannot be used in Processing[20], so for our purposes creating this simple container class to hold these bits of data we need is easiest.

```
package com.catehuston.imagefilter.model;

public class HSBColor {

  public final float h;
  public final float s;
  public final float b;

  public HSBColor(float h, float s, float b) {
    this.h = h;
    this.s = s;
    this.b = b;
  }
}
```

ColorHelper and Associated Tests

`ColorHelper` is where all the image manipulation lives. The methods in this class could be static if not for needing a `PixelColorHelper`. (Although we won't get into the debate about the merits of static methods here.)

```
package com.catehuston.imagefilter.color;

import processing.core.PApplet;

import com.catehuston.imagefilter.model.HSBColor;
import com.catehuston.imagefilter.model.IFAImage;

public class ColorHelper {

  private final PixelColorHelper pixelColorHelper;

  public ColorHelper(PixelColorHelper pixelColorHelper) {
    this.pixelColorHelper = pixelColorHelper;
  }

  public boolean hueInRange(float hue, int hueRange, float lower, float upper) {
    // Need to compensate for it being circular - can go around.
```

[18] https://www.processing.org/reference/color_datatype.html
[19] https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html
[20] http://processing.org/reference/javadoc/core/processing/core/PApplet.html

```
  if (lower < 0) {
    lower += hueRange;
  }
  if (upper > hueRange) {
    upper -= hueRange;
  }
  if (lower < upper) {
    return hue < upper && hue > lower;
  } else {
    return hue < upper || hue > lower;
  }
}

public HSBColor getDominantHue(PApplet applet, IFAImage image, int hueRange) {
  image.loadPixels();
  int numberOfPixels = image.getPixels().length;
  int[] hues = new int[hueRange];
  float[] saturations = new float[hueRange];
  float[] brightnesses = new float[hueRange];

  for (int i = 0; i < numberOfPixels; i++) {
    int pixel = image.getPixel(i);
    int hue = Math.round(pixelColorHelper.hue(applet, pixel));
    float saturation = pixelColorHelper.saturation(applet, pixel);
    float brightness = pixelColorHelper.brightness(applet, pixel);
    hues[hue]++;
    saturations[hue] += saturation;
    brightnesses[hue] += brightness;
  }

  // Find the most common hue.
  int hueCount = hues[0];
  int hue = 0;
  for (int i = 1; i < hues.length; i++) {
    if (hues[i] > hueCount) {
      hueCount = hues[i];
      hue = i;
    }
  }

  // Return the color to display.
  float s = saturations[hue] / hueCount;
  float b = brightnesses[hue] / hueCount;
  return new HSBColor(hue, s, b);
}

public void processImageForHue(PApplet applet, IFAImage image, int hueRange,
    int hueTolerance, boolean showHue) {
  applet.colorMode(PApplet.HSB, (hueRange - 1));
  image.loadPixels();
  int numberOfPixels = image.getPixels().length;
```

```
    HSBColor dominantHue = getDominantHue(applet, image, hueRange);
    // Manipulate photo, grayscale any pixel that isn't close to that hue.
    float lower = dominantHue.h - hueTolerance;
    float upper = dominantHue.h + hueTolerance;
    for (int i = 0; i < numberOfPixels; i++) {
      int pixel = image.getPixel(i);
      float hue = pixelColorHelper.hue(applet, pixel);
      if (hueInRange(hue, hueRange, lower, upper) == showHue) {
        float brightness = pixelColorHelper.brightness(applet, pixel);
        image.setPixel(i, pixelColorHelper.color(applet, brightness));
      }
    }
    image.updatePixels();
  }

  public void applyColorFilter(PApplet applet, IFAImage image, int minRed,
      int minGreen, int minBlue, int colorRange) {
    applet.colorMode(PApplet.RGB, colorRange);
    image.loadPixels();
    int numberOfPixels = image.getPixels().length;
    for (int i = 0; i < numberOfPixels; i++) {
      int pixel = image.getPixel(i);
      float alpha = pixelColorHelper.alpha(applet, pixel);
      float red = pixelColorHelper.red(applet, pixel);
      float green = pixelColorHelper.green(applet, pixel);
      float blue = pixelColorHelper.blue(applet, pixel);

      red = (red >= minRed) ? red : 0;
      green = (green >= minGreen) ? green : 0;
      blue = (blue >= minBlue) ? blue : 0;

      image.setPixel(i, pixelColorHelper.color(applet, red, green, blue, alpha));
    }
  }
}
```

We don't want to test this with whole images, because we want images that we know the properties of and reason about. We approximate this by mocking the images and making them return an array of pixels — in this case, 5. This allows us to verify that the behavior is as expected. Earlier we covered the concept of mock objects, and here we see their use. We are using Mockito[21] as our mock object framework.

To create a mock we use the `@Mock` annotation on an instance variable, and it will be mocked at runtime by the `MockitoJUnitRunner`.

To stub (set the behavior of) a method, we use:

```
when(mock.methodCall()).thenReturn(value)
```

To verify a method was called, we use `verify(mock.methodCall())`.

[21] `http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html`

We'll show a few example test cases here; if you'd like to see the rest, visit the source folder for this project in the *500 Lines or Less* GitHub repository[22].

```
package com.catehuston.imagefilter.color;

/* ... Imports omitted ... */

@RunWith(MockitoJUnitRunner.class)
public class ColorHelperTest {

  @Mock PApplet applet;
  @Mock IFAImage image;
  @Mock PixelColorHelper pixelColorHelper;

  ColorHelper colorHelper;

  private static final int px1 = 1000;
  private static final int px2 = 1010;
  private static final int px3 = 1030;
  private static final int px4 = 1040;
  private static final int px5 = 1050;
  private static final int[] pixels = { px1, px2, px3, px4, px5 };

  @Before public void setUp() throws Exception {
    colorHelper = new ColorHelper(pixelColorHelper);
    when(image.getPixels()).thenReturn(pixels);
    setHsbValuesForPixel(0, px1, 30F, 5F, 10F);
    setHsbValuesForPixel(1, px2, 20F, 6F, 11F);
    setHsbValuesForPixel(2, px3, 30F, 7F, 12F);
    setHsbValuesForPixel(3, px4, 50F, 8F, 13F);
    setHsbValuesForPixel(4, px5, 30F, 9F, 14F);
  }

  private void setHsbValuesForPixel(int px, int color, float h, float s, float b) {
    when(image.getPixel(px)).thenReturn(color);
    when(pixelColorHelper.hue(applet, color)).thenReturn(h);
    when(pixelColorHelper.saturation(applet, color)).thenReturn(s);
    when(pixelColorHelper.brightness(applet, color)).thenReturn(b);
  }

  private void setRgbValuesForPixel(int px, int color, float r, float g, float b,
            float alpha) {
    when(image.getPixel(px)).thenReturn(color);
    when(pixelColorHelper.red(applet, color)).thenReturn(r);
    when(pixelColorHelper.green(applet, color)).thenReturn(g);
    when(pixelColorHelper.blue(applet, color)).thenReturn(b);
    when(pixelColorHelper.alpha(applet, color)).thenReturn(alpha);
  }

    @Test public void testHsbColorFromImage() {
```

[22] https://github.com/aosabook/500lines/tree/master/image-filters

```
    HSBColor color = colorHelper.getDominantHue(applet, image, 100);
    verify(image).loadPixels();

    assertEquals(30F, color.h, 0);
    assertEquals(7F, color.s, 0);
    assertEquals(12F, color.b, 0);
  }

  @Test public void testProcessImageNoHue() {
    when(pixelColorHelper.color(applet, 11F)).thenReturn(11);
    when(pixelColorHelper.color(applet, 13F)).thenReturn(13);
    colorHelper.processImageForHue(applet, image, 60, 2, false);
    verify(applet).colorMode(PApplet.HSB, 59);
    verify(image, times(2)).loadPixels();
    verify(image).setPixel(1, 11);
    verify(image).setPixel(3, 13);
  }

  @Test public void testApplyColorFilter() {
    setRgbValuesForPixel(0, px1, 10F, 12F, 14F, 60F);
    setRgbValuesForPixel(1, px2, 20F, 22F, 24F, 70F);
    setRgbValuesForPixel(2, px3, 30F, 32F, 34F, 80F);
    setRgbValuesForPixel(3, px4, 40F, 42F, 44F, 90F);
    setRgbValuesForPixel(4, px5, 50F, 52F, 54F, 100F);

    when(pixelColorHelper.color(applet, 0F, 0F, 0F, 60F)).thenReturn(5);
    when(pixelColorHelper.color(applet, 20F, 0F, 0F, 70F)).thenReturn(15);
    when(pixelColorHelper.color(applet, 30F, 32F, 0F, 80F)).thenReturn(25);
    when(pixelColorHelper.color(applet, 40F, 42F, 44F, 90F)).thenReturn(35);
    when(pixelColorHelper.color(applet, 50F, 52F, 54F, 100F)).thenReturn(45);

    colorHelper.applyColorFilter(applet, image, 15, 25, 35, 100);
    verify(applet).colorMode(PApplet.RGB, 100);
    verify(image).loadPixels();

    verify(image).setPixel(0, 5);
    verify(image).setPixel(1, 15);
    verify(image).setPixel(2, 25);
    verify(image).setPixel(3, 35);
    verify(image).setPixel(4, 45);
  }
}
```

Notice that:

- We use the `MockitoJUnit` runner.
- We mock `PApplet`, `IFAImage` (created for expressly this purpose), and `ImageColorHelper`.
- Test methods are annotated with `@Test`[23]. If you want to ignore a test (e.g., whilst debugging) you can add the annotation `@Ignore`.
- In `setup()`, we create the pixel array and have the mock image always return it.
- Helper methods make it easier to set expectations for recurring tasks (e.g., `set*ForPixel()`).

Image State and Associated Tests

`ImageState` holds the current "state" of the image — the image itself, and the settings and filters that will be applied. We'll omit the full implementation of `ImageState` here, but we'll show how it can be tested. You can visit the source repository for this project to see the full details.

```
package com.catehuston.imagefilter.model;

import processing.core.PApplet;
import com.catehuston.imagefilter.color.ColorHelper;

public class ImageState {

  enum ColorMode {
    COLOR_FILTER,
    SHOW_DOMINANT_HUE,
    HIDE_DOMINANT_HUE
  }

  private final ColorHelper colorHelper;
  private IFAImage image;
  private String filepath;

  public static final int INITIAL_HUE_TOLERANCE = 5;

  ColorMode colorModeState = ColorMode.COLOR_FILTER;
  int blueFilter = 0;
  int greenFilter = 0;
  int hueTolerance = 0;
  int redFilter = 0;

  public ImageState(ColorHelper colorHelper) {
    this.colorHelper = colorHelper;
    image = new IFAImage();
    hueTolerance = INITIAL_HUE_TOLERANCE;
  }
  /* ... getters & setters */
  public void updateImage(PApplet applet, int hueRange, int rgbColorRange,
          int imageMax) { ... }
```

[23] Method names in tests need not start with `test` as of JUnit 4, but habits are hard to break.

```
  public void processKeyPress(char key, int inc, int rgbColorRange,
          int hueIncrement, int hueRange) { ... }

  public void setUpImage(PApplet applet, int imageMax) { ... }

  public void resetImage(PApplet applet, int imageMax) { ... }

  // For testing purposes only.
  protected void set(IFAImage image, ColorMode colorModeState,
            int redFilter, int greenFilter, int blueFilter, int hueTolerance) { ... }
}
```

Here we can test that the appropriate actions happen for the given state; that fields are incremented and decremented appropriately.

```
package com.catehuston.imagefilter.model;

/* ... Imports omitted ... */

@RunWith(MockitoJUnitRunner.class)
public class ImageStateTest {

  @Mock PApplet applet;
  @Mock ColorHelper colorHelper;
  @Mock IFAImage image;

  private ImageState imageState;

  @Before public void setUp() throws Exception {
    imageState = new ImageState(colorHelper);
  }

  private void assertState(ColorMode colorMode, int redFilter,
      int greenFilter, int blueFilter, int hueTolerance) {
    assertEquals(colorMode, imageState.getColorMode());
    assertEquals(redFilter, imageState.redFilter());
    assertEquals(greenFilter, imageState.greenFilter());
    assertEquals(blueFilter, imageState.blueFilter());
    assertEquals(hueTolerance, imageState.hueTolerance());
  }

  @Test public void testUpdateImageDominantHueHidden() {
    imageState.setFilepath("filepath");
    imageState.set(image, ColorMode.HIDE_DOMINANT_HUE, 5, 10, 15, 10);

    imageState.updateImage(applet, 100, 100, 500);

    verify(image).update(applet, "filepath");
    verify(colorHelper).processImageForHue(applet, image, 100, 10, false);
    verify(colorHelper).applyColorFilter(applet, image, 5, 10, 15, 100);
    verify(image).updatePixels();
  }
```

```
@Test public void testUpdateDominantHueShowing() {
  imageState.setFilepath("filepath");
  imageState.set(image, ColorMode.SHOW_DOMINANT_HUE, 5, 10, 15, 10);

  imageState.updateImage(applet, 100, 100, 500);

  verify(image).update(applet, "filepath");
  verify(colorHelper).processImageForHue(applet, image, 100, 10, true);
  verify(colorHelper).applyColorFilter(applet, image, 5, 10, 15, 100);
  verify(image).updatePixels();
}

@Test public void testUpdateRGBOnly() {
  imageState.setFilepath("filepath");
  imageState.set(image, ColorMode.COLOR_FILTER, 5, 10, 15, 10);

  imageState.updateImage(applet, 100, 100, 500);

  verify(image).update(applet, "filepath");
  verify(colorHelper, never()).processImageForHue(any(PApplet.class),
              any(IFAImage.class), anyInt(), anyInt(), anyBoolean());
  verify(colorHelper).applyColorFilter(applet, image, 5, 10, 15, 100);
  verify(image).updatePixels();
}

@Test public void testKeyPress() {
  imageState.processKeyPress('r', 5, 100, 2, 200);
  assertState(ColorMode.COLOR_FILTER, 5, 0, 0, 5);

  imageState.processKeyPress('e', 5, 100, 2, 200);
  assertState(ColorMode.COLOR_FILTER, 0, 0, 0, 5);

  imageState.processKeyPress('g', 5, 100, 2, 200);
  assertState(ColorMode.COLOR_FILTER, 0, 5, 0, 5);

  imageState.processKeyPress('f', 5, 100, 2, 200);
  assertState(ColorMode.COLOR_FILTER, 0, 0, 0, 5);

  imageState.processKeyPress('b', 5, 100, 2, 200);
  assertState(ColorMode.COLOR_FILTER, 0, 0, 5, 5);

  imageState.processKeyPress('v', 5, 100, 2, 200);
  assertState(ColorMode.COLOR_FILTER, 0, 0, 0, 5);

  imageState.processKeyPress('h', 5, 100, 2, 200);
  assertState(ColorMode.HIDE_DOMINANT_HUE, 0, 0, 0, 5);

  imageState.processKeyPress('i', 5, 100, 2, 200);
  assertState(ColorMode.HIDE_DOMINANT_HUE, 0, 0, 0, 7);
```

```
    imageState.processKeyPress('u', 5, 100, 2, 200);
    assertState(ColorMode.HIDE_DOMINANT_HUE, 0, 0, 0, 5);

    imageState.processKeyPress('h', 5, 100, 2, 200);
    assertState(ColorMode.COLOR_FILTER, 0, 0, 0, 5);

    imageState.processKeyPress('s', 5, 100, 2, 200);
    assertState(ColorMode.SHOW_DOMINANT_HUE, 0, 0, 0, 5);

    imageState.processKeyPress('s', 5, 100, 2, 200);
    assertState(ColorMode.COLOR_FILTER, 0, 0, 0, 5);

    // Random key should do nothing.
    imageState.processKeyPress('z', 5, 100, 2, 200);
    assertState(ColorMode.COLOR_FILTER, 0, 0, 0, 5);
  }

  @Test public void testSave() {
    imageState.set(image, ColorMode.SHOW_DOMINANT_HUE, 5, 10, 15, 10);
    imageState.setFilepath("filepath");
    imageState.processKeyPress('w', 5, 100, 2, 200);

    verify(image).save("filepath-new.png");
  }

  @Test public void testSetupImageLandscape() {
    imageState.set(image, ColorMode.SHOW_DOMINANT_HUE, 5, 10, 15, 10);
    when(image.getWidth()).thenReturn(20);
    when(image.getHeight()).thenReturn(8);
    imageState.setUpImage(applet, 10);
    verify(image).update(applet, null);
    verify(image).resize(10, 4);
  }

  @Test public void testSetupImagePortrait() {
    imageState.set(image, ColorMode.SHOW_DOMINANT_HUE, 5, 10, 15, 10);
    when(image.getWidth()).thenReturn(8);
    when(image.getHeight()).thenReturn(20);
    imageState.setUpImage(applet, 10);
    verify(image).update(applet, null);
    verify(image).resize(4, 10);
  }

  @Test public void testResetImage() {
    imageState.set(image, ColorMode.SHOW_DOMINANT_HUE, 5, 10, 15, 10);
    imageState.resetImage(applet, 10);
    assertState(ColorMode.COLOR_FILTER, 0, 0, 0, 5);
  }
}
```

Notice that:

- We exposed a protected initialization method `set` for testing that helps us quickly get the system under test into a specific state.
- We mock `PApplet`, `ColorHelper`, and `IFAImage` (created expressly for this purpose).
- This time we use a helper (`assertState()`) to simplify asserting the state of the image.

Measuring test coverage

I use EclEmma[24] to measure test coverage within Eclipse. Overall for the app we have 81% test coverage, with none of `ImageFilterApp` covered, 94.8% for `ImageState`, and 100% for `ColorHelper`.

ImageFilterApp

This is where everything is tied together, but we want as little as possible here. The App is hard to unit test (much of it is layout), but because we've pushed so much of the app's functionality into our own tested classes, we're able to assure ourselves that the important parts are working as intended.

We set the size of the app, and do the layout. (These things are verified by running the app and making sure it looks okay — no matter how good the test coverage, this step should not be skipped!)

```
package com.catehuston.imagefilter.app;

import java.io.File;

import processing.core.PApplet;

import com.catehuston.imagefilter.color.ColorHelper;
import com.catehuston.imagefilter.color.PixelColorHelper;
import com.catehuston.imagefilter.model.ImageState;

@SuppressWarnings("serial")
public class ImageFilterApp extends PApplet {

  static final String INSTRUCTIONS = "...";

  static final int FILTER_HEIGHT = 2;
  static final int FILTER_INCREMENT = 5;
  static final int HUE_INCREMENT = 2;
  static final int HUE_RANGE = 100;
  static final int IMAGE_MAX = 640;
  static final int RGB_COLOR_RANGE = 100;
  static final int SIDE_BAR_PADDING = 10;
  static final int SIDE_BAR_WIDTH = RGB_COLOR_RANGE + 2 * SIDE_BAR_PADDING + 50;

  private ImageState imageState;

  boolean redrawImage = true;

  @Override
```

[24] http://www.eclemma.org/installation.html\#marketplace

```
public void setup() {
  noLoop();
  imageState = new ImageState(new ColorHelper(new PixelColorHelper()));

  // Set up the view.
  size(IMAGE_MAX + SIDE_BAR_WIDTH, IMAGE_MAX);
  background(0);

  chooseFile();
}

@Override
public void draw() {
  // Draw image.
  if (imageState.image().image() != null && redrawImage) {
    background(0);
    drawImage();
  }

  colorMode(RGB, RGB_COLOR_RANGE);
  fill(0);
  rect(IMAGE_MAX, 0, SIDE_BAR_WIDTH, IMAGE_MAX);
  stroke(RGB_COLOR_RANGE);
  line(IMAGE_MAX, 0, IMAGE_MAX, IMAGE_MAX);

  // Draw red line
  int x = IMAGE_MAX + SIDE_BAR_PADDING;
  int y = 2 * SIDE_BAR_PADDING;
  stroke(RGB_COLOR_RANGE, 0, 0);
  line(x, y, x + RGB_COLOR_RANGE, y);
  line(x + imageState.redFilter(), y - FILTER_HEIGHT,
      x + imageState.redFilter(), y + FILTER_HEIGHT);

  // Draw green line
  y += 2 * SIDE_BAR_PADDING;
  stroke(0, RGB_COLOR_RANGE, 0);
  line(x, y, x + RGB_COLOR_RANGE, y);
  line(x + imageState.greenFilter(), y - FILTER_HEIGHT,
      x + imageState.greenFilter(), y + FILTER_HEIGHT);

  // Draw blue line
  y += 2 * SIDE_BAR_PADDING;
  stroke(0, 0, RGB_COLOR_RANGE);
  line(x, y, x + RGB_COLOR_RANGE, y);
  line(x + imageState.blueFilter(), y - FILTER_HEIGHT,
      x + imageState.blueFilter(), y + FILTER_HEIGHT);

  // Draw white line.
  y += 2 * SIDE_BAR_PADDING;
  stroke(HUE_RANGE);
  line(x, y, x + 100, y);
```

```
  line(x + imageState.hueTolerance(), y - FILTER_HEIGHT,
      x + imageState.hueTolerance(), y + FILTER_HEIGHT);

  y += 4 * SIDE_BAR_PADDING;
  fill(RGB_COLOR_RANGE);
  text(INSTRUCTIONS, x, y);
  updatePixels();
}

// Callback for selectInput(), has to be public to be found.
public void fileSelected(File file) {
  if (file == null) {
    println("User hit cancel.");
  } else {
    imageState.setFilepath(file.getAbsolutePath());
    imageState.setUpImage(this, IMAGE_MAX);
    redrawImage = true;
    redraw();
  }
}

private void drawImage() {
  imageMode(CENTER);
  imageState.updateImage(this, HUE_RANGE, RGB_COLOR_RANGE, IMAGE_MAX);
  image(imageState.image().image(), IMAGE_MAX/2, IMAGE_MAX/2,
              imageState.image().getWidth(), imageState.image().getHeight());
  redrawImage = false;
}

@Override
public void keyPressed() {
  switch(key) {
  case 'c':
    chooseFile();
    break;
  case 'p':
    redrawImage = true;
    break;
  case ' ':
    imageState.resetImage(this, IMAGE_MAX);
    redrawImage = true;
    break;
  }
  imageState.processKeyPress(key, FILTER_INCREMENT, RGB_COLOR_RANGE,
              HUE_INCREMENT, HUE_RANGE);
  redraw();
}

private void chooseFile() {
  // Choose the file.
  selectInput("Select a file to process:", "fileSelected");
```

```
  }
}
```

Notice that:

- Our implementation extends `PApplet`.
- Most work is done in `ImageState`.
- `fileSelected()` is the callback for `selectInput()`.
- `static final` constants are defined up at the top.

11.9 The Value of Prototyping

In real world programming, we spend a lot of time on productionisation work. Making things look just so. Maintaining 99.9% uptime. We spend more time on corner cases than refining algorithms.

These constraints and requirements are important for our users. However there's also space for freeing ourselves from them to play and explore.

Eventually, I decided to port this to a native mobile app. Processing has an Android library, but as many mobile developers do, I opted to go iOS first. I had years of iOS experience, although I'd done little with CoreGraphics, but I don't think even if I had had this idea initially, I would have been able to build it straight away on iOS. The platform forced me to operate in the RGB color space, and made it hard to extract the pixels from the image (hello, C). Memory and waiting was a major risk.

There were exhilarating moments, when it worked for the first time. When it first ran on my device… without crashing. When I optimized memory usage by 66% and cut seconds off the runtime. And there were large periods of time locked away in a dark room, cursing intermittently.

Because I had my prototype, I could explain to my business partner and our designer what I was thinking and what the app would do. It meant I deeply understood how it would work, and it was just a question of making it work nicely on this other platform. I knew what I was aiming for, so at the end of a long day shut away fighting with it and feeling like I had little to show for it I kept going… and hit an exhilarating moment and milestone the following morning.

So, how do you find the dominant color in an image? There's an app for that: Show & Hide[25].

[25] `http://showandhide.com`

A Simple Object Model

Carl Friedrich Bolz

14.1 Introduction

Object-oriented programming is one of the major programming paradigms in use today, with a lot of languages providing some form of object-orientation. While on the surface the mechanisms that different object-oriented programming languages provide to the programmer are very similar, the details can vary a lot. Commonalities of most languages are the presence of objects and some kind of inheritance mechanism. Classes, however, are a feature that not every language supports directly. For example, in prototype-based languages like Self or JavaScript, the concept of class does not exist and objects instead inherit directly from each other.

Understanding the differences between different object models can be interesting. They often reveal the family resemblance between different languages. It can be useful to put the model of a new language into the context of the models of other languages, both to quickly understand the new model, and to get a better feeling for the programming language design space.

This chapter explores the implementation of a series of very simple object models. It starts out with simple instances and classes, and the ability to call methods on instances. This is the "classical" object-oriented approach that was established in early OO languages such as Simula 67 and Smalltalk. This model is then extended step by step, the next two steps exploring different language design choices, and the last step improving the efficiency of the object model. The final model is not that of a real language, but an idealized, simplified version of Python's object model.

The object models presented in this chapter will be implemented in Python. The code works on both Python 2.7 and 3.4. To understand the behaviour and the design choices better, the chapter will also present tests for the object model. The tests can be run with either py.test or nose.

The choice of Python as an implementation language is quite unrealistic. A "real" VM is typically implemented in a low-level language like C/C++ and needs a lot of attention to engineering detail to make it efficient. However, the simpler implementation language makes it easier to focus on actual behaviour differences instead of getting bogged down by implementation details.

14.2 Method-Based Model

The object model we will start out with is an extremely simplified version of that of Smalltalk. Smalltalk was an object-oriented programming language designed by Alan Kay's group at Xerox PARC in the 1970s. It popularized object-oriented programming, and is the source of many features

found in today's programming languages. One of the core tenets of Smalltalk's language design was "everything is an object". Smalltalk's most immediate successor in use today is Ruby, which uses a more C-like syntax but retains most of Smalltalk's object model.

The object model in this section will have classes and instances of them, the ability to read and write attributes into objects, the ability to call methods on objects, and the ability for a class to be a subclass of another class. Right from the beginning, classes will be completely ordinary objects that can themselves have attributes and methods.

A note on terminology: In this chapter I will use the word "instance" to mean -"an object that is not a class".

A good approach to start with is to write a test to specify what the to-be-implemented behaviour should be. All tests presented in this chapter will consist of two parts. First, a bit of regular Python code defining and using a few classes, and making use of increasingly advanced features of the Python object model. Second, the corresponding test using the object model we will implement in this chapter, instead of normal Python classes.

The mapping between using normal Python classes and using our object model will be done manually in the tests. For example, instead of writing `obj.attribute` in Python, in the object model we would use a method `obj.read_attr("attribute")`. This mapping would, in a real language implementation, be done by the interpreter of the language, or a compiler.

A further simplification in this chapter is that we make no sharp distinction between the code that implements the object model and the code that is used to write the methods used in the objects. In a real system, the two would often be implemented in different programming languages.

Let us start with a simple test for reading and writing object fields.

```
def test_read_write_field():
    # Python code
    class A(object):
        pass
    obj = A()
    obj.a = 1
    assert obj.a == 1

    obj.b = 5
    assert obj.a == 1
    assert obj.b == 5

    obj.a = 2
    assert obj.a == 2
    assert obj.b == 5

    # Object model code
    A = Class(name="A", base_class=OBJECT, fields={}, metaclass=TYPE)
    obj = Instance(A)
    obj.write_attr("a", 1)
    assert obj.read_attr("a") == 1

    obj.write_attr("b", 5)
    assert obj.read_attr("a") == 1
    assert obj.read_attr("b") == 5

    obj.write_attr("a", 2)
```

```
    assert obj.read_attr("a") == 2
    assert obj.read_attr("b") == 5
```

The test uses three things that we have to implement. The classes `Class` and `Instance` represent classes and instances of our object model, respectively. There are two special instances of class: `OBJECT` and `TYPE`. `OBJECT` corresponds to `object` in Python and is the ultimate base class of the inheritance hierarchy. `TYPE` corresponds to `type` in Python and is the type of all classes.

To do anything with instances of `Class` and `Instance`, they implement a shared interface by inheriting from a shared base class `Base` that exposes a number of methods:

```
class Base(object):
    """ The base class that all of the object model classes inherit from. """

    def __init__(self, cls, fields):
        """ Every object has a class. """
        self.cls = cls
        self._fields = fields

    def read_attr(self, fieldname):
        """ read field 'fieldname' out of the object """
        return self._read_dict(fieldname)

    def write_attr(self, fieldname, value):
        """ write field 'fieldname' into the object """
        self._write_dict(fieldname, value)

    def isinstance(self, cls):
        """ return True if the object is an instance of class cls """
        return self.cls.issubclass(cls)

    def callmethod(self, methname, *args):
        """ call method 'methname' with arguments 'args' on object """
        meth = self.cls._read_from_class(methname)
        return meth(self, *args)

    def _read_dict(self, fieldname):
        """ read an field 'fieldname' out of the object's dict """
        return self._fields.get(fieldname, MISSING)

    def _write_dict(self, fieldname, value):
        """ write a field 'fieldname' into the object's dict """
        self._fields[fieldname] = value

MISSING = object()
```

The `Base` class implements storing the class of an object, and a dictionary containing the field values of the object. Now we need to implement `Class` and `Instance`. The constructor of `Instance` takes the class to be instantiated and initializes the `fields` `dict` as an empty dictionary. Otherwise `Instance` is just a very thin subclass around `Base` that does not add any extra functionality.

The constructor of `Class` takes the name of the class, the base class, the dictionary of the class and the metaclass. For classes, the fields are passed into the constructor by the user of the object

model. The class constructor also takes a base class, which the tests so far don't need but which we will make use of in the next section.

```
class Instance(Base):
    """Instance of a user-defined class. """

    def __init__(self, cls):
        assert isinstance(cls, Class)
        Base.__init__(self, cls, {})


class Class(Base):
    """ A User-defined class. """

    def __init__(self, name, base_class, fields, metaclass):
        Base.__init__(self, metaclass, fields)
        self.name = name
        self.base_class = base_class
```

Since classes are also a kind of object, they (indirectly) inherit from `Base`. Thus, the class needs to be an instance of another class: its metaclass.

Now our first test almost passes. The only missing bit is the definition of the base classes `TYPE` and `OBJECT`, which are both instances of `Class`. For these we will make a major departure from the Smalltalk model, which has a fairly complex metaclass system. Instead we will use the model introduced in ObjVlisp[1], which Python adopted.

In the ObjVlisp model, `OBJECT` and `TYPE` are intertwined. `OBJECT` is the base class of all classes, meaning it has no base class. `TYPE` is a subclass of `OBJECT`. By default, every class is an instance of `TYPE`. In particular, both `TYPE` and `OBJECT` are instances of `TYPE`. However, the programmer can also subclass `TYPE` to make a new metaclass:

```
# set up the base hierarchy as in Python (the ObjVLisp model)
# the ultimate base class is OBJECT
OBJECT = Class(name="object", base_class=None, fields={}, metaclass=None)
# TYPE is a subclass of OBJECT
TYPE = Class(name="type", base_class=OBJECT, fields={}, metaclass=None)
# TYPE is an instance of itself
TYPE.cls = TYPE
# OBJECT is an instance of TYPE
OBJECT.cls = TYPE
```

To define new metaclasses, it is enough to subclass `TYPE`. However, in the rest of this chapter we won't do that; we'll simply always use `TYPE` as the metaclass of every class.

Now the first test passes. The second test checks that reading and writing attributes works on classes as well. It's easy to write, and passes immediately.

[1] P. Cointe, "Metaclasses are first class: The ObjVlisp Model," SIGPLAN Not, vol. 22, no. 12, pp. 156–162, 1987.

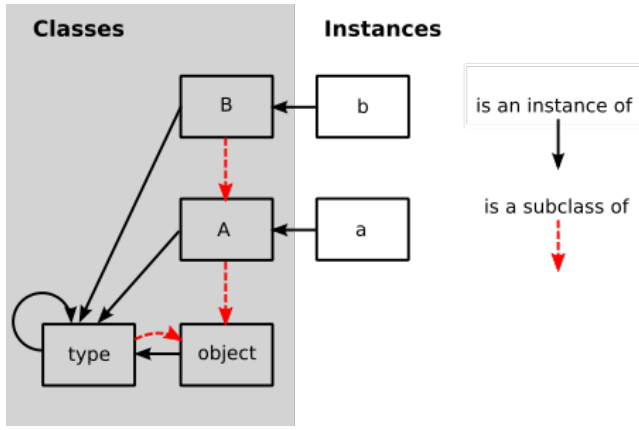

Figure 14.1: Inheritance

```
def test_read_write_field_class():
    # classes are objects too
    # Python code
    class A(object):
        pass
    A.a = 1
    assert A.a == 1
    A.a = 6
    assert A.a == 6

    # Object model code
    A = Class(name="A", base_class=OBJECT, fields={"a": 1}, metaclass=TYPE)
    assert A.read_attr("a") == 1
    A.write_attr("a", 5)
    assert A.read_attr("a") == 5
```

`isinstance` Checkingisinstance Checking

So far we haven't taken advantage of the fact that objects have classes. The next test implements the `isinstance` machinery:

```
def test_isinstance():
    # Python code
    class A(object):
        pass
    class B(A):
        pass
    b = B()
    assert isinstance(b, B)
    assert isinstance(b, A)
    assert isinstance(b, object)
    assert not isinstance(b, type)

    # Object model code
```

```
    A = Class(name="A", base_class=OBJECT, fields={}, metaclass=TYPE)
    B = Class(name="B", base_class=A, fields={}, metaclass=TYPE)
    b = Instance(B)
    assert b.isinstance(B)
    assert b.isinstance(A)
    assert b.isinstance(OBJECT)
    assert not b.isinstance(TYPE)
```

To check whether an object `obj` is an instance of a certain class `cls`, it is enough to check whether `cls` is a superclass of the class of `obj`, or the class itself. To check whether a class is a superclass of another class, the chain of superclasses of that class is walked. If and only if the other class is found in that chain, it is a superclass. The chain of superclasses of a class, including the class itself, is called the "method resolution order" of that class. It can easily be computed recursively:

```
class Class(Base):
    ...

    def method_resolution_order(self):
        """ compute the method resolution order of the class """
        if self.base_class is None:
            return [self]
        else:
            return [self] + self.base_class.method_resolution_order()

    def issubclass(self, cls):
        """ is self a subclass of cls? """
        return cls in self.method_resolution_order()
```

With that code, the test passes.

Calling Methods

The remaining missing feature for this first version of the object model is the ability to call methods on objects. In this chapter we will implement a simple single inheritance model.

```
def test_callmethod_simple():
    # Python code
    class A(object):
        def f(self):
            return self.x + 1
    obj = A()
    obj.x = 1
    assert obj.f() == 2

    class B(A):
        pass
    obj = B()
    obj.x = 1
    assert obj.f() == 2 # works on subclass too

    # Object model code
```

```
    def f_A(self):
        return self.read_attr("x") + 1
    A = Class(name="A", base_class=OBJECT, fields={"f": f_A}, metaclass=TYPE)
    obj = Instance(A)
    obj.write_attr("x", 1)
    assert obj.callmethod("f") == 2

    B = Class(name="B", base_class=A, fields={}, metaclass=TYPE)
    obj = Instance(B)
    obj.write_attr("x", 2)
    assert obj.callmethod("f") == 3
```

To find the correct implementation of a method that is sent to an object, we walk the method resolution order of the class of the object. The first method found in the dictionary of one of the classes in the method resolution order is called:

```
class Class(Base):
    ...

    def _read_from_class(self, methname):
        for cls in self.method_resolution_order():
            if methname in cls._fields:
                return cls._fields[methname]
        return MISSING
```

Together with the code for `callmethod` in the `Base` implementation, this passes the test.

To make sure that methods with arguments work as well, and that overriding of methods is implemented correctly, we can use the following slightly more complex test, which already passes:

```
def test_callmethod_subclassing_and_arguments():
    # Python code
    class A(object):
        def g(self, arg):
            return self.x + arg
    obj = A()
    obj.x = 1
    assert obj.g(4) == 5

    class B(A):
        def g(self, arg):
            return self.x + arg * 2
    obj = B()
    obj.x = 4
    assert obj.g(4) == 12

    # Object model code
    def g_A(self, arg):
        return self.read_attr("x") + arg
    A = Class(name="A", base_class=OBJECT, fields={"g": g_A}, metaclass=TYPE)
    obj = Instance(A)
    obj.write_attr("x", 1)
```

```
    assert obj.callmethod("g", 4) == 5

    def g_B(self, arg):
        return self.read_attr("x") + arg * 2
    B = Class(name="B", base_class=A, fields={"g": g_B}, metaclass=TYPE)
    obj = Instance(B)
    obj.write_attr("x", 4)
    assert obj.callmethod("g", 4) == 12
```

14.3 Attribute-Based Model

Now that the simplest version of our object model is working, we can think of ways to change it. This section will introduce the distinction between a method-based model and an attribute-based model. This is one of the core differences between Smalltalk, Ruby, and JavaScript on the one hand and Python and Lua on the other hand.

The method-based model has the method-calling as the primitive of program execution:

```
result = obj.f(arg1, arg2)
```

The attribute-based model splits up method calling into two steps: looking up an attribute and calling the result:

```
method = obj.f
result = method(arg1, arg2)
```

This difference can be shown in the following test:

```
def test_bound_method():
    # Python code
    class A(object):
        def f(self, a):
            return self.x + a + 1
    obj = A()
    obj.x = 2
    m = obj.f
    assert m(4) == 7

    class B(A):
        pass
    obj = B()
    obj.x = 1
    m = obj.f
    assert m(10) == 12 # works on subclass too

    # Object model code
    def f_A(self, a):
        return self.read_attr("x") + a + 1
    A = Class(name="A", base_class=OBJECT, fields={"f": f_A}, metaclass=TYPE)
    obj = Instance(A)
    obj.write_attr("x", 2)
```

```
    m = obj.read_attr("f")
    assert m(4) == 7

    B = Class(name="B", base_class=A, fields={}, metaclass=TYPE)
    obj = Instance(B)
    obj.write_attr("x", 1)
    m = obj.read_attr("f")
    assert m(10) == 12
```

While the setup is the same as the corresponding test for method calls, the way that the methods are called is different. First, the attribute with the name of the method is looked up on the object. The result of that lookup operation is a *bound method*, an object that encapsulates both the object as well as the function found in the class. Next, that bound method is called with a call operation[2].

To implement this behaviour, we need to change the `Base.read_attr` implementation. If the attribute is not found in the dictionary, it is looked for in the class. If it is found in the class, and the attribute is a callable, it needs to be turned into a bound method. To emulate a bound method we simply use a closure. In addition to changing `Base.read_attr` we can also change `Base.callmethod` to use the new approach to calling methods to make sure all the tests still pass.

```
class Base(object):
    ...
    def read_attr(self, fieldname):
        """ read field 'fieldname' out of the object """
        result = self._read_dict(fieldname)
        if result is not MISSING:
            return result
        result = self.cls._read_from_class(fieldname)
        if _is_bindable(result):
            return _make_boundmethod(result, self)
        if result is not MISSING:
            return result
        raise AttributeError(fieldname)

    def callmethod(self, methname, *args):
        """ call method 'methname' with arguments 'args' on object """
        meth = self.read_attr(methname)
        return meth(*args)

def _is_bindable(meth):
    return callable(meth)

def _make_boundmethod(meth, self):
    def bound(*args):
        return meth(self, *args)
    return bound
```

The rest of the code does not need to be changed at all.

[2]It seems that the attribute-based model is conceptually more complex, because it needs both method lookup and call. In practice, calling something is defined by looking up and calling a special attribute `__call__`, so conceptual simplicity is regained. This won't be implemented in this chapter, however.)

14.4 Meta-Object Protocols

In addition to "normal" methods that are called directly by the program, many dynamic languages support *special methods*. These are methods that aren't meant to be called directly but will be called by the object system. In Python those special methods usually have names that start and end with two underscores; e.g., `__init__`. Special methods can be used to override primitive operations and provide custom behaviour for them instead. Thus, they are hooks that tell the object model machinery exactly how to do certain things. Python's object model has dozens of special methods[3].

Meta-object protocols were introduced by Smalltalk, but were used even more by the object systems for Common Lisp, such as CLOS. That is also where the name *meta-object protocol*, for collections of special methods, was coined[4].

In this chapter we will add three such meta-hooks to our object model. They are used to fine-tune what exactly happens when reading and writing attributes. The special methods we will add first are `__getattr__` and `__setattr__`, which closely follow the behaviour of Python's namesakes.

Customizing Reading and Writing and Attribute

The method `__getattr__` is called by the object model when the attribute that is being looked up is not found by normal means; i.e., neither on the instance nor on the class. It gets the name of the attribute being looked up as an argument. An equivalent of the `__getattr__` special method was part of early Smalltalk[5] systems under the name `doesNotUnderstand:`.

The case of `__setattr__` is a bit different. Since setting an attribute always creates it, `__setattr__` is always called when setting an attribute. To make sure that a `__setattr__` method always exists, the `OBJECT` class has a definition of `__setattr__`. This base implementation simply does what setting an attribute did so far, which is write the attribute into the object's dictionary. This also makes it possible for a user-defined `__setattr__` to delegate to the base `OBJECT.__setattr__` in some cases.

A test for these two special methods is the following:

```
def test_getattr():
    # Python code
    class A(object):
        def __getattr__(self, name):
            if name == "fahrenheit":
                return self.celsius * 9. / 5. + 32
            raise AttributeError(name)

        def __setattr__(self, name, value):
            if name == "fahrenheit":
                self.celsius = (value - 32) * 5. / 9.
            else:
                # call the base implementation
                object.__setattr__(self, name, value)
    obj = A()
    obj.celsius = 30
```

[3] `https://docs.python.org/2/reference/datamodel.html\#special-method-names`

[4] G. Kiczales, J. des Rivieres, and D. G. Bobrow, The Art of the Metaobject Protocol. Cambridge, Mass: The MIT Press, 1991.

[5] A. Goldberg, Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983, page 61.

```
    assert obj.fahrenheit == 86 # test __getattr__
    obj.celsius = 40
    assert obj.fahrenheit == 104

    obj.fahrenheit = 86 # test __setattr__
    assert obj.celsius == 30
    assert obj.fahrenheit == 86

    # Object model code
    def __getattr__(self, name):
        if name == "fahrenheit":
            return self.read_attr("celsius") * 9. / 5. + 32
        raise AttributeError(name)
    def __setattr__(self, name, value):
        if name == "fahrenheit":
            self.write_attr("celsius", (value - 32) * 5. / 9.)
        else:
            # call the base implementation
            OBJECT.read_attr("__setattr__")(self, name, value)

    A = Class(name="A", base_class=OBJECT,
              fields={"__getattr__": __getattr__, "__setattr__": __setattr__},
              metaclass=TYPE)
    obj = Instance(A)
    obj.write_attr("celsius", 30)
    assert obj.read_attr("fahrenheit") == 86 # test __getattr__
    obj.write_attr("celsius", 40)
    assert obj.read_attr("fahrenheit") == 104
    obj.write_attr("fahrenheit", 86) # test __setattr__
    assert obj.read_attr("celsius") == 30
    assert obj.read_attr("fahrenheit") == 86
```

To pass these tests, the `Base.read_attr` and `Base.write_attr` methods need to be changed:

```
class Base(object):
    ...

    def read_attr(self, fieldname):
        """ read field 'fieldname' out of the object """
        result = self._read_dict(fieldname)
        if result is not MISSING:
            return result
        result = self.cls._read_from_class(fieldname)
        if _is_bindable(result):
            return _make_boundmethod(result, self)
        if result is not MISSING:
            return result
        meth = self.cls._read_from_class("__getattr__")
        if meth is not MISSING:
            return meth(self, fieldname)
        raise AttributeError(fieldname)
```

```
    def write_attr(self, fieldname, value):
        """ write field 'fieldname' into the object """
        meth = self.cls._read_from_class("__setattr__")
        return meth(self, fieldname, value)
```

The procedure for reading an attribute is changed to call the `__getattr__` method with the fieldname as an argument, if the method exists, instead of raising an error. Note that `__getattr__` (and indeed all special methods in Python) is looked up on the class only, instead of recursively calling `self.read_attr("__getattr__")`. That is because the latter would lead to an infinite recursion of `read_attr` if `__getattr__` were not defined on the object.

Writing of attributes is fully deferred to the `__setattr__` method. To make this work, `OBJECT` needs to have a `__setattr__` method that calls the default behaviour, as follows:

```
def OBJECT__setattr__(self, fieldname, value):
    self._write_dict(fieldname, value)
OBJECT = Class("object", None, {"__setattr__": OBJECT__setattr__}, None)
```

The behaviour of `OBJECT__setattr__` is like the previous behaviour of `write_attr`. With these modifications, the new test passes.

Descriptor Protocol

The above test to provide automatic conversion between different temperature scales worked but was annoying to write, as the attribute name needed to be checked explicitly in the `__getattr__` and `__setattr__` methods. To get around this, the *descriptor protocol* was introduced in Python.

While `__getattr__` and `__setattr__` are called on the object the attribute is being read from, the descriptor protocol calls a special method on the *result* of getting an attribute from an object. It can be seen as the generalization of binding a method to an object – and indeed, binding a method to an object is done using the descriptor protocol. In addition to bound methods, the most important use case for the descriptor protocol in Python is the implementation of `staticmethod`, `classmethod` and `property`.

In this subsection we will introduce the subset of the descriptor protocol which deals with binding objects. This is done using the special method `__get__`, and is best explained with an example test:

```
def test_get():
    # Python code
    class FahrenheitGetter(object):
        def __get__(self, inst, cls):
            return inst.celsius * 9. / 5. + 32

    class A(object):
        fahrenheit = FahrenheitGetter()
    obj = A()
    obj.celsius = 30
    assert obj.fahrenheit == 86

    # Object model code
    class FahrenheitGetter(object):
        def __get__(self, inst, cls):
            return inst.read_attr("celsius") * 9. / 5. + 32
```

```
    A = Class(name="A", base_class=OBJECT,
              fields={"fahrenheit": FahrenheitGetter()},
              metaclass=TYPE)
    obj = Instance(A)
    obj.write_attr("celsius", 30)
    assert obj.read_attr("fahrenheit") == 86
```

The `__get__` method is called on the `FahrenheitGetter` instance after that has been looked up in the class of `obj`. The arguments to `__get__` are the instance where the lookup was done[6].

Implementing this behaviour is easy. We simply need to change `_is_bindable` and `_make_boundmethod`:

```
def _is_bindable(meth):
    return hasattr(meth, "__get__")

def _make_boundmethod(meth, self):
    return meth.__get__(self, None)
```

This makes the test pass. The previous tests about bound methods also still pass, as Python's functions have a `__get__` method that returns a bound method object.

In practice, the descriptor protocol is quite a lot more complex. It also supports `__set__` to override what setting an attribute means on a per-attribute basis. Also, the current implementation is cutting a few corners. Note that `_make_boundmethod` calls the method `__get__` on the implementation level, instead of using `meth.read_attr("__get__")`. This is necessary since our object model borrows functions and thus methods from Python, instead of having a representation for them that uses the object model. A more complete object model would have to solve this problem.

14.5 Instance Optimization

While the first three variants of the object model were concerned with behavioural variation, in this last section we will look at an optimization without any behavioural impact. This optimization is called *maps* and was pioneered in the VM for the Self programming language[7]. It is still one of the most important object model optimizations: it's used in PyPy and all modern JavaScript VMs, such as V8 (where the optimization is called *hidden classes*).

The optimization starts from the following observation: In the object model as implemented so far all instances use a full dictionary to store their attributes. A dictionary is implemented using a hash map, which takes a lot of memory. In addition, the dictionaries of instances of the same class typically have the same keys as well. For example, given a class `Point`, the keys of all its instances' dictionaries are likely `"x"` and `"y"`.

The maps optimization exploits this fact. It effectively splits up the dictionary of every instance into two parts. A part storing the keys (the map) that can be shared between all instances with the same set of attribute names. The instance then only stores a reference to the shared map and the values of the attributes in a list (which is a lot more compact in memory than a dictionary). The map stores a mapping from attribute names to indexes into that list.

A simple test of that behaviour looks like this:

[6] In Python the second argument is the class where the attribute was found, though we will ignore that here.

[7] C. Chambers, D. Ungar, and E. Lee, "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes," in OOPSLA, 1989, vol. 24.

```
def test_maps():
    # white box test inspecting the implementation
    Point = Class(name="Point", base_class=OBJECT, fields={}, metaclass=TYPE)
    p1 = Instance(Point)
    p1.write_attr("x", 1)
    p1.write_attr("y", 2)
    assert p1.storage == [1, 2]
    assert p1.map.attrs == {"x": 0, "y": 1}

    p2 = Instance(Point)
    p2.write_attr("x", 5)
    p2.write_attr("y", 6)
    assert p1.map is p2.map
    assert p2.storage == [5, 6]

    p1.write_attr("x", -1)
    p1.write_attr("y", -2)
    assert p1.map is p2.map
    assert p1.storage == [-1, -2]

    p3 = Instance(Point)
    p3.write_attr("x", 100)
    p3.write_attr("z", -343)
    assert p3.map is not p1.map
    assert p3.map.attrs == {"x": 0, "z": 1}
```

Note that this is a different flavour of test than the ones we've written before. All previous tests just tested the behaviour of the classes via the exposed interfaces. This test instead checks the implementation details of the `Instance` class by reading internal attributes and comparing them to predefined values. Therefore this test can be called a *white-box* test.

The `attrs` attribute of the map of `p1` describes the layout of the instance as having two attributes `"x"` and `"y"` which are stored at position 0 and 1 of the `storage` of `p1`. Making a second instance `p2` and adding to it the same attributes in the same order will make it end up with the same map. If, on the other hand, a different attribute is added, the map can of course not be shared.

The `Map` class looks like this:

```
class Map(object):
    def __init__(self, attrs):
        self.attrs = attrs
        self.next_maps = {}

    def get_index(self, fieldname):
        return self.attrs.get(fieldname, -1)

    def next_map(self, fieldname):
        assert fieldname not in self.attrs
        if fieldname in self.next_maps:
            return self.next_maps[fieldname]
        attrs = self.attrs.copy()
        attrs[fieldname] = len(attrs)
        result = self.next_maps[fieldname] = Map(attrs)
```

```
        return result

EMPTY_MAP = Map({})
```

Maps have two methods, `get_index` and `next_map`. The former is used to find the index of an attribute name in the object's storage. The latter is used when a new attribute is added to an object. In that case the object needs to use a different map, which `next_map` computes. The method uses the `next_maps` dictionary to cache already created maps. That way, objects that have the same layout also end up using the same `Map` object.

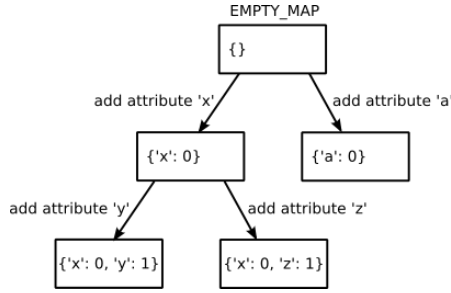

Figure 14.2: Map transitions

The `Instance` implementation that uses maps looks like this:

```
class Instance(Base):
    """Instance of a user-defined class. """

    def __init__(self, cls):
        assert isinstance(cls, Class)
        Base.__init__(self, cls, None)
        self.map = EMPTY_MAP
        self.storage = []

    def _read_dict(self, fieldname):
        index = self.map.get_index(fieldname)
        if index == -1:
            return MISSING
        return self.storage[index]

    def _write_dict(self, fieldname, value):
        index = self.map.get_index(fieldname)
        if index != -1:
            self.storage[index] = value
        else:
            new_map = self.map.next_map(fieldname)
            self.storage.append(value)
            self.map = new_map
```

The class now passes `None` as the fields dictionary to `Base`, as `Instance` will store the content of the dictionary in another way. Therefore it needs to override the `_read_dict` and `_write_dict` methods. In a real implementation, we would refactor the `Base` class so that it is no longer responsible for storing the fields dictionary, but for now having instances store `None` there is good enough.

A newly created instance starts out using the `EMPTY_MAP`, which has no attributes, and empty storage. To implement `_read_dict`, the instance's map is asked for the index of the attribute name. Then the corresponding entry of the storage list is returned.

Writing into the fields dictionary has two cases. On the one hand the value of an existing attribute can be changed. This is done by simply changing the storage at the corresponding index. On the other hand, if the attribute does not exist yet, a *map transition* (Figure 14.2) is needed using the `next_map` method. The value of the new attribute is appended to the storage list.

What does this optimization achieve? It optimizes use of memory in the common case where there are many instances with the same layout. It is not a universal optimization: code that creates instances with wildly different sets of attributes will have a larger memory footprint than if we just use dictionaries.

This is a common problem when optimizing dynamic languages. It is often not possible to find optimizations that are faster or use less memory in all cases. In practice, the optimizations chosen apply to how the language is *typically* used, while potentially making behaviour worse for programs that use extremely dynamic features.

Another interesting aspect of maps is that, while here they only optimize for memory use, in actual VMs that use a just-in-time (JIT) compiler they also improve the performance of the program. To achieve that, the JIT uses the maps to compile attribute lookups to a lookup in the objects' storage at a fixed offset, getting rid of all dictionary lookups completely[8].

14.6 Potential Extensions

It is easy to extend our object model and experiment with various language design choices. Here are some possibilities:

- The easiest thing to do is to add further special methods. Some easy and interesting ones to add are `__init__`, `__getattribute__`, `__set__`.
- The model can be very easily extended to support multiple inheritance. To do this, every class would get a list of base classes. Then the `Class.method_resolution_order` method would need to be changed to support looking up methods. A simple method resolution order could be computed using a depth-first search with removal of duplicates. A more complicated but better one is the C3 algorithm[9], which adds better handling in the base of diamond-shaped multiple inheritance hierarchies and rejects insensible inheritance patterns.
- A more radical change is to switch to a prototype model, which involves the removal of the distinction between classes and instances.

14.7 Conclusions

Some of the core aspects of the design of an object-oriented programming language are the details of its object model. Writing small object model prototypes is an easy and fun way to understand the inner workings of existing languages better and to get insights into the design space of object-oriented

[8] How that works is beyond the scope of this chapter. I tried to give a reasonably readable account of it in a paper I wrote a few years ago. It uses an object model that is basically a variant of the one in this chapter: C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo, "Runtime feedback in a meta-tracing JIT for efficient dynamic languages," in Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, New York, NY, USA, 2011, pp. 9:1–9:8.

[9] `https://www.python.org/download/releases/2.3/mro/`

languages. Playing with object models is a good way to experiment with different language design ideas without having to worry about the more boring parts of language implementation, such as parsing and executing code.

Such object models can also be useful in practice, not just as vehicles for experimentation. They can be embedded in and used from other languages. Examples of this approach are common: the GObject object model, written in C, that's used in GLib and other Gnome libraries; or the various class system implementations in JavaScript.

Optical Character Recognition (OCR)

Marina Samuel

15.1 Introduction

What if your computer could wash your dishes, do your laundry, cook you dinner, and clean your home? I think I can safely say that most people would be happy to get a helping hand! But what would it take for a computer to be able to perform these tasks in the same way that humans can?

The famous computer scientist Alan Turing proposed the Turing Test as a way to identify whether a machine could have intelligence indistinguishable from that of a human being. The test involves a human posing questions to two hidden entities, one human, and the other a machine, and trying to identify which is which. If the interrogator is unable to identify the machine, then the machine is considered to have human-level intelligence.

While there is a lot of controversy surrounding whether the Turing Test is a valid assessment of intelligence, and whether we can build such intelligent machines, there is no doubt that machines with some degree of intelligence already exist. There is currently software that helps robots navigate an office and perform small tasks, or help those suffering with Alzheimer's. More common examples of Artificial Intelligence (A.I.) are the way that Google estimates what you're looking for when you search for some keywords, or the way that Facebook decides what to put in your news feed.

One well known application of A.I. is Optical Character Recognition (OCR). An OCR system is a piece of software that can take images of handwritten characters as input and interpret them into machine readable text. While you may not think twice when depositing a handwritten cheque into a bank machine , there is some interesting work going on in the background. This chapter will examine a working example of a simple OCR system that recognizes numerical digits using an Artificial Neural Network (ANN). But first, let's establish a bit more context.

15.2 What is Artificial Intelligence?

While Turing's definition of intelligence sounds reasonable, at the end of the day what constitutes intelligence is fundamentally a philosophical debate. Computer scientists have, however, categorized certain types of systems and algorithms into branches of AI. Each branch is used to solve certain sets of problems. These branches include the following examples, as well as many others[1]:

[1] http://www-formal.stanford.edu/jmc/whatisai/node2.html

- Logical and probabilistic deduction and inference based on some predefined knowledge of a world. e.g. Fuzzy inference[2] can help a thermostat decide when to turn on the air conditioning when it detects that the temperature is hot and the atmosphere is humid
- Heuristic search. e.g. Searching can be used to find the best possible next move in a game of chess by searching all possible moves and choosing the one that most improves your position
- Machine learning (ML) with feedback models. e.g. Pattern-recognition problems like OCR.

In general, ML involves using large data sets to train a system to identify patterns. The training data sets may be labelled, meaning the system's expected outputs are specified for given inputs, or unlabelled meaning expected outputs are not specified. Algorithms that train systems with unlabelled data are called *unsupervised* algorithms and those that train with labelled data are called *supervised*. Many ML algorithms and techniques exist for creating OCR systems, of which ANNs are one approach.

15.3 Artificial Neural Networks

What Are ANNs?

An ANN is a structure consisting of interconnected nodes that communicate with one another. The structure and its functionality are inspired by neural networks found in a biological brain. Hebbian Theory[3] explains how these networks can learn to identify patterns by physically altering their structure and link strengths. Similarly, a typical ANN (shown in Figure 15.1) has connections between nodes that have a weight which is updated as the network learns. The nodes labelled "+1" are called *biases*. The leftmost blue column of nodes are *input nodes*, the middle column contains *hidden nodes*, and the rightmost column contains *output nodes*. There may be many columns of hidden nodes, known as *hidden layers*.

The values inside all of the circular nodes in Figure 15.1 represent the output of the node. If we call the output of the nth node from the top in layer L as a $n(L)$ and the connection between the ith node in layer L and the jth node in layer $L+1$ as $w_j^{(L)}i$, then the output of node $a_2^{(2)}$ is:

$$a_2^{(2)} = f(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)})$$

where $f(.)$ is known as the *activation function* and b is the *bias*. An activation function is the decision-maker for what type of output a node has. A bias is an additional node with a fixed output of 1 that may be added to an ANN to improve its accuracy. We'll see more details on both of these in Section 15.4.

This type of network topology is called a *feedforward* neural network because there are no cycles in the network. ANNs with nodes whose outputs feed into their inputs are called recurrent neural networks. There are many algorithms that can be applied to train feedforward ANNs; one commonly used algorithm is called *backpropagation*. The OCR system we will implement in this chapter will use backpropagation.

How Do We Use ANNs?

Like most other ML approaches, the first step for using backpropagation is to decide how to transform or reduce our problem into one that can be solved by an ANN. In other words, how can we manipulate

[2] http://www.cs.princeton.edu/courses/archive/fall07/cos436/HIDDEN/Knapp/fuzzy004.htm
[3] http://www.nbb.cornell.edu/neurobio/linster/BioNB420/hebb.pdf

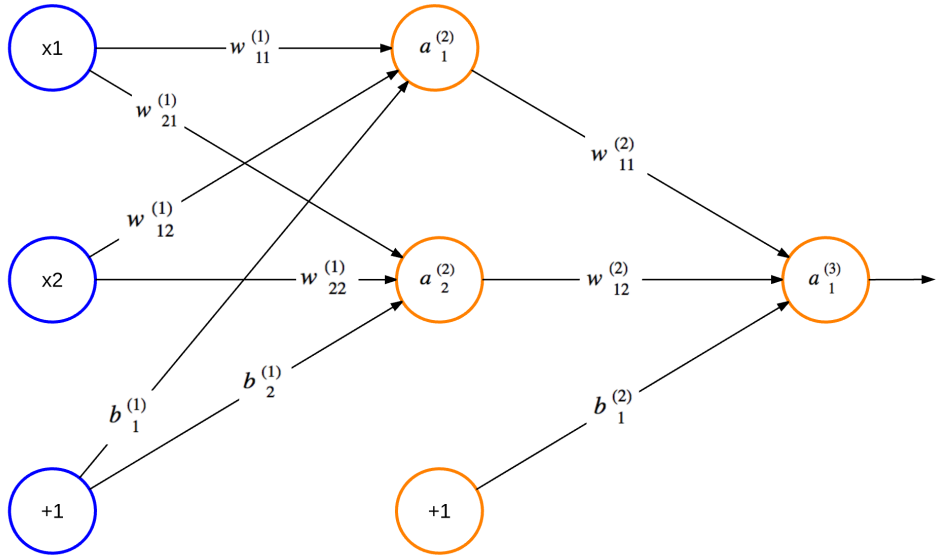

Figure 15.1: An Artificial Neural Network

our input data so we can feed it into the ANN? For the case of our OCR system, we can use the positions of the pixels for a given digit as input. It is worth noting that, often times, choosing the input data format is not this simple. If we were analyzing large images to identify shapes in them, for instance, we may need to pre-process the image to identify contours within it. These contours would be the input.

Once we've decided on our input data format, what's next? Since backpropagation is a supervised algorithm, it will need to be trained with labelled data, as mentioned in Section 15.2. Thus, when passing the pixel positions as training input, we must also pass the associated digit. This means that we must find or gather a large data set of drawn digits and associated values.

The next step is to partition the data set into a training set and validation set. The training data is used to run the backpropagation algorithm to set the weights of the ANN. The validation data is used to make predictions using the trained network and compute its accuracy. If we were comparing the performance of backpropagation vs. another algorithm on our data, we would split the data[4] into 50% for training, 25% for comparing performance of the 2 algorithms (validation set) and the final 25% for testing accuracy of the chosen algorithm (test set). Since we're not comparing algorithms, we can group one of the 25% sets as part of the training set and use 75% of the data to train the network and 25% for validating that it was trained well.

The purpose of identifying the accuracy of the ANN is two-fold. First, it is to avoid the problem of *overfitting*. Overfitting occurs when the network has a much higher accuracy on predicting the training set than the validation set. Overfitting tells us that the chosen training data does not generalize well enough and needs to be refined. Secondly, testing the accuracy of several different numbers of hidden layers and hidden nodes helps in designing the most optimal ANN size. An optimal ANN size will have enough hidden nodes and layers to make accurate predictions but also as few nodes/connections as possible to reduce computational overhead that may slow down training and

[4] `http://www-group.slac.stanford.edu/sluo/Lectures/stat_lecture_files/sluo2006lec7.pdf`

predictions. Once the optimal size has been decided and the network has been trained, it's ready to make predictions!

15.4 Design Decisions in a Simple OCR System

In the last few paragraphs we've gone over some of the basics of feedforward ANNs and how to use them. Now it's time to talk about how we can build an OCR system.

First off, we must decide what we want our system to be able to do. To keep things simple, let's allow users to draw a single digit and be able to train the OCR system with that drawn digit or to request that the system predict what the drawn digit is. While an OCR system could run locally on a single machine, having a client-server setup gives much more flexibility. It makes crowd-sourced training of an ANN possible and allows powerful servers to handle intensive computations.

Our OCR system will consist of 5 main components, divided into 5 files. There will be:

- a client (`ocr.js`)
- a server (`server.py`)
- a simple user interface (`ocr.html`)
- an ANN trained via backpropagation (`ocr.py`)
- an ANN design script (`neural_network_design.py`)

The user interface will be simple: a canvas to draw digits on and buttons to either train the ANN or request a prediction. The client will gather the drawn digit, translate it into an array, and pass it to the server to be processed either as a training sample or as a prediction request. The server will simply route the training or prediction request by making API calls to the ANN module. The ANN module will train the network with an existing data set on its first initialization. It will then save the ANN weights to a file and re-load them on subsequent startups. This module is where the core of training and prediction logic happens. Finally, the design script is for experimenting with different hidden node counts and deciding what works best. Together, these pieces give us a very simplistic, but functional OCR system.

Now that we've thought about how the system will work at a high level, it's time to put the concepts into code!

A Simple Interface (`ocr.html`)A Simple Interface (ocr.html)

As mentioned earlier, the first step is to gather data for training the network. We could upload a sequence of hand-written digits to the server, but that would be awkward. Instead, we could have users actually handwrite the digits on the page using an HTML canvas. We could then give them a couple of options to either train or test the network, where training the network also involves specifying what digit was drawn. This way it is possible to easily outsource the data collection by pointing people to a website to receive their input. Here's some HTML to get us started.

```
<html>
<head>
    <script src="ocr.js"></script>
    <link rel="stylesheet" type="text/css" href="ocr.css">
</head>
<body onload="ocrDemo.onLoadFunction()">
    <div id="main-container" style="text-align: center;">
        <h1>OCR Demo</h1>
```

```
        <canvas id="canvas" width="200" height="200"></canvas>
        <form name="input">
            <p>Digit: <input id="digit" type="text"> </p>
            <input type="button" value="Train" onclick="ocrDemo.train()">
            <input type="button" value="Test" onclick="ocrDemo.test()">
            <input type="button" value="Reset" onclick="ocrDemo.resetCanvas();"/>
        </form>
    </div>
</body>
</html>
```

An OCR Client (`ocr.js`)An OCR Client (ocr.js)

Since a single pixel on an HTML canvas might be hard to see, we can represent a single pixel for the ANN input as a square of 10x10 real pixels. Thus the real canvas is 200x200 pixels and it is represented by a 20x20 canvas from the perspective of the ANN. The variables below will help us keep track of these measurements.

```
var ocrDemo = {
    CANVAS_WIDTH: 200,
    TRANSLATED_WIDTH: 20,
    PIXEL_WIDTH: 10, // TRANSLATED_WIDTH = CANVAS_WIDTH / PIXEL_WIDTH
```

We can then outline the pixels in the new representation so they are easier to see. Here we have a blue grid generated by `drawGrid()`.

```
    drawGrid: function(ctx) {
        for (var x = this.PIXEL_WIDTH, y = this.PIXEL_WIDTH;
                 x < this.CANVAS_WIDTH; x += this.PIXEL_WIDTH,
                 y += this.PIXEL_WIDTH) {
            ctx.strokeStyle = this.BLUE;
            ctx.beginPath();
            ctx.moveTo(x, 0);
            ctx.lineTo(x, this.CANVAS_WIDTH);
            ctx.stroke();

            ctx.beginPath();
            ctx.moveTo(0, y);
            ctx.lineTo(this.CANVAS_WIDTH, y);
            ctx.stroke();
        }
    },
```

We also need to store the data drawn on the grid in a form that can be sent to the server. For simplicity, we can have an array called `data` which labels an uncoloured, black pixel as `0` and a coloured white pixel as `1`. We also need some mouse listeners on the canvas so we know when to call `fillSquare()` to colour a pixel white while a user is drawing a digit. These listeners should keep track of whether we are in a drawing state and then call `fillSquare()` to do some simple math and decide which pixels need to be filled in.

```
onMouseMove: function(e, ctx, canvas) {
    if (!canvas.isDrawing) {
        return;
    }
    this.fillSquare(ctx,
        e.clientX - canvas.offsetLeft, e.clientY - canvas.offsetTop);
},

onMouseDown: function(e, ctx, canvas) {
    canvas.isDrawing = true;
    this.fillSquare(ctx,
        e.clientX - canvas.offsetLeft, e.clientY - canvas.offsetTop);
},

onMouseUp: function(e) {
    canvas.isDrawing = false;
},

fillSquare: function(ctx, x, y) {
    var xPixel = Math.floor(x / this.PIXEL_WIDTH);
    var yPixel = Math.floor(y / this.PIXEL_WIDTH);
    this.data[((xPixel - 1)  * this.TRANSLATED_WIDTH + yPixel) - 1] = 1;

    ctx.fillStyle = '#ffffff';
    ctx.fillRect(xPixel * this.PIXEL_WIDTH, yPixel * this.PIXEL_WIDTH,
        this.PIXEL_WIDTH, this.PIXEL_WIDTH);
},
```

Now we're getting closer to the juicy stuff! We need a function that prepares training data to be sent to the server. Here we have a relatively straight forward `train()` function that does some error checking on the data to be sent, adds it to `trainArray` and sends it off by calling `sendData()`.

```
train: function() {
    var digitVal = document.getElementById("digit").value;
    if (!digitVal || this.data.indexOf(1) < 0) {
        alert("Please type and draw a digit value in order to train the network");
        return;
    }
    this.trainArray.push({"y0": this.data, "label": parseInt(digitVal)});
    this.trainingRequestCount++;

    // Time to send a training batch to the server.
    if (this.trainingRequestCount == this.BATCH_SIZE) {
        alert("Sending training data to server...");
        var json = {
            trainArray: this.trainArray,
            train: true
        };

        this.sendData(json);
        this.trainingRequestCount = 0;
        this.trainArray = [];
```

```
    }
},
```

An interesting design worth noting here is the use of `trainingRequestCount`, `trainArray`, and `BATCH_SIZE`. What's happening here is that `BATCH_SIZE` is some pre-defined constant for how much training data a client will keep track of before it sends a batched request to the server to be processed by the OCR. The main reason to batch requests is to avoid overwhelming the server with many requests at once. If many clients exist (e.g. many users are on the `ocr.html` page training the system), or if another layer existed in the client that takes scanned drawn digits and translated them to pixels to train the network, a `BATCH_SIZE` of 1 would result in many, unnecessary requests. This approach is good because it gives more flexibility to the client, however, in practice, batching should also take place on the server, when needed. A denial of service (DoS) attack could occur in which a malicious client purposely sends many requests to the server to overwhelm it so that it breaks down.

We will also need a `test()` function. Similar to `train()`, it should do a simple check on the validity of the data and send it off. For `test()`, however, no batching occurs since users should be able to request a prediction and get immediate results.

```
test: function() {
    if (this.data.indexOf(1) < 0) {
        alert("Please draw a digit in order to test the network");
        return;
    }
    var json = {
        image: this.data,
        predict: true
    };
    this.sendData(json);
},
```

Finally, we will need some functions to make an HTTP POST request, receive a response, and handle any potential errors along the way.

```
receiveResponse: function(xmlHttp) {
    if (xmlHttp.status != 200) {
        alert("Server returned status " + xmlHttp.status);
        return;
    }
    var responseJSON = JSON.parse(xmlHttp.responseText);
    if (xmlHttp.responseText && responseJSON.type == "test") {
        alert("The neural network predicts you wrote a \'"
               + responseJSON.result + '\'');
    }
},

onError: function(e) {
    alert("Error occurred while connecting to server: " + e.target.statusText);
},

sendData: function(json) {
    var xmlHttp = new XMLHttpRequest();
    xmlHttp.open('POST', this.HOST + ":" + this.PORT, false);
```

```
    xmlHttp.onload = function() { this.receiveResponse(xmlHttp); }.bind(this);
    xmlHttp.onerror = function() { this.onError(xmlHttp) }.bind(this);
    var msg = JSON.stringify(json);
    xmlHttp.setRequestHeader('Content-length', msg.length);
    xmlHttp.setRequestHeader("Connection", "close");
    xmlHttp.send(msg);
}
```

A Server (`server.py`)A Server (server.py)

Despite being a small server that simply relays information, we still need to consider how to receive and handle the HTTP requests. First we need to decide what kind of HTTP request to use. In the last section, the client is using POST, but why did we decide on this? Since data is being sent to the server, a PUT or POST request makes the most sense. We only need to send a json body and no URL parameters. So in theory, a GET request could have worked as well but would not make sense semantically. The choice between PUT and POST, however, is a long, on-going debate among programmers; KNPLabs summarizes the issues with humour[5].

Another consideration is whether to send the "train" vs. "predict" requests to different endpoints (e.g. `http://localhost/train` and `http://localhost/predict`) or the same endpoint which then processes the data separately. In this case, we can go with the latter approach since the difference between what is done with the data in each case is minor enough to fit into a short if statement. In practice, it would be better to have these as separate endpoints if the server were to do any more detailed processing for each request type. This decision, in turn impacted what server error codes were used when. For example, a 400 "Bad Request" error is sent when neither "train" or "predict" is specified in the payload. If separate endpoints were used instead, this would not be an issue. The processing done in the background by the OCR system may fail for any reason and if it's not handled correctly within the server, a 500 "Internal Server Error" is sent. Again, if the endpoints were separated, there would have been more room to go into detail to send more appropriate errors. For example, identifying that an internal server error was actually caused by a bad request.

Finally, we need to decide when and where to initialize the OCR system. A good approach would be to initialize it within `server.py` but before the server is started. This is because on first run, the OCR system needs to train the network on some pre-existing data the first time it starts and this may take a few minutes. If the server started before this processing was complete, any requests to train or predict would throw an exception since the OCR object would not yet have been initialized, given the current implementation. Another possible implementation could create some inaccurate initial ANN to be used for the first few queries while the new ANN is asynchronously trained in the background. This alternative approach does allow the ANN to be used immediately, but the implementation is more complex and it would only save on time on server startup if the servers are reset. This type of implementation would be more beneficial for an OCR service that requires high availability.

Here we have the majority of our server code in one short function that handles POST requests.

```
def do_POST(s):
    response_code = 200
    response = ""
    var_len = int(s.headers.get('Content-Length'))
    content = s.rfile.read(var_len);
    payload = json.loads(content);
```

[5] `https://knpuniversity.com/screencast/rest/put-versus-post`

```
if payload.get('train'):
    nn.train(payload['trainArray'])
    nn.save()
elif payload.get('predict'):
    try:
        response = {
            "type":"test",
            "result":nn.predict(str(payload['image']))
        }
    except:
        response_code = 500
else:
    response_code = 400

s.send_response(response_code)
s.send_header("Content-type", "application/json")
s.send_header("Access-Control-Allow-Origin", "*")
s.end_headers()
if response:
    s.wfile.write(json.dumps(response))
return
```

Designing a Feedforward ANN (`neural_network_design.py`)Designing a Feedforward ANN (neural_network_design.py)

When designing a feedforward ANN, there are a few factors we must consider. The first is what activation function to use. We mentioned activation functions earlier as the decision-maker for a node's output. The type of the decision an activation function makes will help us decide which one to use. In our case, we will be designing an ANN that outputs a value between 0 and 1 for each digit (0-9). Values closer to 1 would mean the ANN predicts this is the drawn digit and values closer to 0 would mean it's predicted to not be the drawn digit. Thus, we want an activation function that would have outputs either close to 0 or close to 1. We also need a function that is differentiable because we will need the derivative for our backpropagation computation. A commonly used function in this case is the sigmoid because it satisfies both these constraints. StatSoft provides a nice list[6] of common activation functions and their properties.

A second factor to consider is whether we want to include biases. We've mentioned biases a couple of times before but haven't really talked about what they are or why we use them. Let's try to understand this by going back to how the output of a node is computed in Figure 15.1. Suppose we had a single input node and a single output node, our output formula would be $y = f(wx)$, where y is the output, $f()$ is the activation function, w is the weight for the link between the nodes, and x is the variable input for the node. The bias is essentially a node whose output is always 1. This would change the output formula to $y = f(wx + b)$ where b is the weight of the connection between the bias node and the next node. If we consider w and b as constants and x as a variable, then adding a bias adds a constant to our linear function input to $f(.)$.

Adding the bias therefore allows for a shift in the y-intercept and in general gives more flexibility for the output of a node. It's often good practice to include biases, especially for ANNs with a small

[6] `http://www.fmi.uni-sofia.bg/fmi/statist/education/textbook/eng/glosa.html`

number of inputs and outputs. Biases allow for more flexibility in the output of the ANN and thus provide the ANN with more room for accuracy. Without biases, we're less likely to make correct predictions with our ANN or would need more hidden nodes to make more accurate predictions.

Other factors to consider are the number of hidden layers and the number of hidden nodes per layer. For larger ANNs with many inputs and outputs, these numbers are decided by trying different values and testing the network's performance. In this case, the performance is measured by training an ANN of a given size and seeing what percentage of the validation set is classified correctly. In most cases, a single hidden layer is sufficient for decent performance, so we only experiment with the number of hidden nodes here.

```
# Try various number of hidden nodes and see what performs best
for i in xrange(5, 50, 5):
    nn = OCRNeuralNetwork(i, data_matrix, data_labels, train_indices, False)
    performance = str(test(data_matrix, data_labels, test_indices, nn))
    print "{i} Hidden Nodes: {val}".format(i=i, val=performance)
```

Here we initialize an ANN with between 5 to 50 hidden nodes in increments of 5. We then call the `test()` function.

```
def test(data_matrix, data_labels, test_indices, nn):
    avg_sum = 0
    for j in xrange(100):
        correct_guess_count = 0
        for i in test_indices:
            test = data_matrix[i]
            prediction = nn.predict(test)
            if data_labels[i] == prediction:
                correct_guess_count += 1

        avg_sum += (correct_guess_count / float(len(test_indices)))
    return avg_sum / 100
```

The inner loop is counting the number of correct classifications which are then divided by the number of attempted classifications at the end. This gives a ratio or percentage accuracy for the ANN. Since each time an ANN is trained, its weights may be slightly different, we repeat this process 100 times in the outer loop so we can take an average of this particular ANN configuration's accuracy. In our case, a sample run of `neural_network_design.py` looks like the following:

```
PERFORMANCE
-----------
5 Hidden Nodes: 0.7792
10 Hidden Nodes: 0.8704
15 Hidden Nodes: 0.8808
20 Hidden Nodes: 0.8864
25 Hidden Nodes: 0.8808
30 Hidden Nodes: 0.888
35 Hidden Nodes: 0.8904
40 Hidden Nodes: 0.8896
45 Hidden Nodes: 0.8928
```

From this output we can conclude that 15 hidden nodes would be most optimal. Adding 5 nodes from 10 to 15 gets us ~1% more accuracy, whereas improving the accuracy by another 1% would require adding another 20 nodes. Increasing the hidden node count also increases computational overhead. So it would take networks with more hidden nodes longer to be trained and to make predictions. Thus we choose to use the last hidden node count that resulted in a dramatic increase in accuracy. Of course, it's possible when designing an ANN that computational overhead is no problem and it's top priority to have the most accurate ANN possible. In that case it would be better to choose 45 hidden nodes instead of 15.

Core OCR Functionality

In this section we'll talk about how the actual training occurs via backpropagation, how we can use the network to make predictions, and other key design decisions for core functionality.

Training via Backpropagation (`ocr.py`)Training via Backpropagation (ocr.py)

We use the backpropagation algorithm to train our ANN. It consists of 4 main steps that are repeated for every sample in the training set, updating the ANN weights each time.

First, we initialize the weights to small (between -1 and 1) random values. In our case, we initialize them to values between -0.06 and 0.06 and store them in matrices `theta1`, `theta2`, `input_layer_bias`, and `hidden_layer_bias`. Since every node in a layer links to every node in the next layer we can create a matrix that has m rows and n columns where n is the number of nodes in one layer and m is the number of nodes in the adjacent layer. This matrix would represent all the weights for the links between these two layers. Here theta1 has 400 columns for our 20x20 pixel inputs and `num_hidden_nodes` rows. Likewise, `theta2` represents the links between the hidden layer and output layer. It has `num_hidden_nodes` columns and `NUM_DIGITS` (`10`) rows. The other two vectors (1 row), `input_layer_bias` and `hidden_layer_bias` represent the biases.

```
def _rand_initialize_weights(self, size_in, size_out):
    return [((x * 0.12) - 0.06) for x in np.random.rand(size_out, size_in)]

        self.theta1 = self._rand_initialize_weights(400, num_hidden_nodes)
        self.theta2 = self._rand_initialize_weights(num_hidden_nodes, 10)
        self.input_layer_bias = self._rand_initialize_weights(1,
                                                              num_hidden_nodes)
        self.hidden_layer_bias = self._rand_initialize_weights(1, 10)
```

The second step is *forward propagation*, which is essentially computing the node outputs as described in Section 15.3, layer by layer starting from the input nodes. Here, `y0` is an array of size 400 with the inputs we wish to use to train the ANN. We multiply `theta1` by `y0` transposed so that we have two matrices with sizes `(num_hidden_nodes x 400) * (400 x 1)` and have a resulting vector of outputs for the hidden layer of size num_hidden_nodes. We then add the bias vector and apply the vectorized sigmoid activation function to this output vector, giving us `y1`. `y1` is the output vector of our hidden layer. The same process is repeated again to compute `y2` for the output nodes. `y2` is now our output layer vector with values representing the likelihood that their index is the drawn number. For example if someone draws an 8, the value of `y2` at the 8th index will be the largest if the ANN has made the correct prediction. However, 6 may have a higher likelihood than 1 of being the drawn digit since it looks more similar to 8 and is more likely to use up the same pixels to be drawn as the 8. `y2` becomes more accurate with each additional drawn digit the ANN is trained with.

```
# The sigmoid activation function. Operates on scalars.
def _sigmoid_scalar(self, z):
    return 1 / (1 + math.e ** -z)
```

```
        y1 = np.dot(np.mat(self.theta1), np.mat(data['y0']).T)
        sum1 =  y1 + np.mat(self.input_layer_bias) # Add the bias
        y1 = self.sigmoid(sum1)

        y2 = np.dot(np.array(self.theta2), y1)
        y2 = np.add(y2, self.hidden_layer_bias) # Add the bias
        y2 = self.sigmoid(y2)
```

The third step is *back propagation*, which involves computing the errors at the output nodes then at every intermediate layer back towards the input. Here we start by creating an expected output vector, `actual_vals`, with a `1` at the index of the digit that represents the value of the drawn digit and `0`s otherwise. The vector of errors at the output nodes, `output_errors`, is computed by subtracting the actual output vector, `y2`, from `actual_vals`. For every hidden layer afterwards, we compute two components. First, we have the next layer's transposed weight matrix multiplied by its output errors. Then we have the derivative of the activation function applied to the previous layer. We then perform an element-wise multiplication on these two components, giving a vector of errors for a hidden layer. Here we call this `hidden_errors`.

```
        actual_vals = [0] * 10
        actual_vals[data['label']] = 1
        output_errors = np.mat(actual_vals).T - np.mat(y2)
        hidden_errors = np.multiply(np.dot(np.mat(self.theta2).T, output_errors),
                                    self.sigmoid_prime(sum1))
```

Weight updates that adjust the ANN weights based on the errors computed earlier. Weights are updated at each layer via matrix multiplication. The error matrix at each layer is multiplied by the output matrix of the previous layer. This product is then multiplied by a scalar called the learning rate and added to the weight matrix. The learning rate is a value between 0 and 1 that influences the speed and accuracy of learning in the ANN. Larger learning rate values will generate an ANN that learns quickly but is less accurate, while smaller values will will generate an ANN that learns slower but is more accurate. In our case, we have a relatively small value for learning rate, 0.1. This works well since we do not need the ANN to be immediately trained in order for a user to continue making train or predict requests. Biases are updated by simply multiplying the learning rate by the layer's error vector.

```
        self.theta1 += self.LEARNING_RATE * np.dot(np.mat(hidden_errors),
                                                   np.mat(data['y0']))
        self.theta2 += self.LEARNING_RATE * np.dot(np.mat(output_errors),
                                                   np.mat(y1).T)
        self.hidden_layer_bias += self.LEARNING_RATE * output_errors
        self.input_layer_bias += self.LEARNING_RATE * hidden_errors
```

Testing a Trained Network (`ocr.py`)Testing a Trained Network (ocr.py)

Once an ANN has been trained via backpropagation, it is fairly straightforward to use it for making predictions. As we can see here, we start by computing the output of the ANN, `y2`, exactly the way

we did in step 2 of backpropagation. Then we look for the index in the vector with the maximum value. This index is the digit predicted by the ANN.

```
def predict(self, test):
    y1 = np.dot(np.mat(self.theta1), np.mat(test).T)
    y1 =  y1 + np.mat(self.input_layer_bias) # Add the bias
    y1 = self.sigmoid(y1)

    y2 = np.dot(np.array(self.theta2), y1)
    y2 = np.add(y2, self.hidden_layer_bias) # Add the bias
    y2 = self.sigmoid(y2)

    results = y2.T.tolist()[0]
    return results.index(max(results))
```

Other Design Decisions (ocr.py)Other Design Decisions (ocr.py)

Many resources are available online that go into greater detail on the implementation of backpropagation. One good resource is from a course by the University of Willamette[7]. It goes over the steps of backpropagation and then explains how it can be translated into matrix form. While the amount of computation using matrices is the same as using loops, the benefit is that the code is simpler and easier to read with fewer nested loops. As we can see, the entire training process is written in under 25 lines of code using matrix algebra.

As mentioned in the introduction of Section 15.4, persisting the weights of the ANN means we do not lose the progress made in training it when the server is shut down or abruptly goes down for any reason. We persist the weights by writing them as JSON to a file. On startup, the OCR loads the ANN's saved weights to memory. The save function is not called internally by the OCR but is up to the server to decide when to perform a save. In our case, the server saves the weights after each update. This is a quick and simple solution but it is not optimal since writing to disk is time consuming. This also prevents us from handling multiple concurrent requests since there is no mechanism to prevent simultaneous writes to the same file. In a more sophisticated server, saves could perhaps be done on shutdown or once every few minutes with some form of locking or a timestamp protocol to ensure no data loss.

```
def save(self):
    if not self._use_file:
        return

    json_neural_network = {
        "theta1":[np_mat.tolist()[0] for np_mat in self.theta1],
        "theta2":[np_mat.tolist()[0] for np_mat in self.theta2],
        "b1":self.input_layer_bias[0].tolist()[0],
        "b2":self.hidden_layer_bias[0].tolist()[0]
    };
    with open(OCRNeuralNetwork.NN_FILE_PATH,'w') as nnFile:
        json.dump(json_neural_network, nnFile)

def _load(self):
```

[7] http://www.willamette.edu/~gorr/classes/cs449/backprop.html

```
if not self._use_file:
    return

with open(OCRNeuralNetwork.NN_FILE_PATH) as nnFile:
    nn = json.load(nnFile)
self.theta1 = [np.array(li) for li in nn['theta1']]
self.theta2 = [np.array(li) for li in nn['theta2']]
self.input_layer_bias = [np.array(nn['b1'][0])]
self.hidden_layer_bias = [np.array(nn['b2'][0])]
```

15.5 Conclusion

Now that we've learned about AI, ANNs, backpropagation, and building an end-to-end OCR system, let's recap the highlights of this chapter and the big picture.

We started off the chapter by giving background on AI, ANNs, and roughly what we will be implementing. We discussed what AI is and examples of how it's used. We saw that AI is essentially a set of algorithms or problem-solving approaches that can provide an answer to a question in a similar manner as a human would. We then took a look at the structure of a Feedforward ANN. We learned that computing the output at a given node was as simple as summing the products of the outputs of the previous nodes and their connecting weights. We talked about how to use an ANN by first formatting the input and partitioning the data into training and validation sets.

Once we had some background, we started talking about creating a web-based, client-server system that would handle user requests to train or test the OCR. We then discussed how the client would interpret the drawn pixels into an array and perform an HTTP request to the OCR server to perform the training or testing. We discussed how our simple server read requests and how to design an ANN by testing performance of several hidden node counts. We finished off by going through the core training and testing code for backpropagation.

Although we've built a seemingly functional OCR system, this chapter simply scratches the surface of how a real OCR system might work. More sophisticated OCR systems could have pre-processed inputs, use hybrid ML algorithms, have more extensive design phases, or other further optimizations.

The Same-Origin Policy

Eunsuk Kang, Santiago Perez De Rosso, and Daniel Jackson

17.1 Introduction

The same-origin policy (SOP) is an important part of the security mechanism of every modern browser. It controls when scripts running in a browser can communicate with one another (roughly, when they originate from the same website). First introduced in Netscape Navigator, the SOP now plays a critical role in the security of web applications; without it, it would be far easier for a malicious hacker to peruse your private photos on Facebook, read your email, or empty your bank account.

But the SOP is far from perfect. At times, it is too restrictive; there are cases (such as mashups) in which scripts from different origins should be able to share a resource but cannot. At other times it is not restrictive enough, leaving corner cases that can be exploited using common attacks such as cross-site request forgery. Furthermore, the design of the SOP has evolved organically over the years and puzzles many developers.

The goal of this chapter is to capture the essence of this important—yet often misunderstood—feature. In particular, we will attempt to answer the following questions:

- Why is the SOP necessary? What are the types of security violations that it prevents?
- How is the behavior of a web application affected by the SOP?
- What are different mechanisms for bypassing the SOP?
- How secure are these mechanisms? What are potential security issues that they introduce?

Covering the SOP in its entirety is a daunting task, given the complexity of the parts that are involved—web servers, browsers, HTTP, HTML documents, client-side scripts, and so on. We would likely get bogged down by the gritty details of all these parts (and consume our 500 lines before even reaching SOP). But how can we hope to be precise without representing crucial details?

17.2 Modeling with Alloy

This chapter is somewhat different from others in this book. Instead of building a working implementation, we will construct an executable model that serves as a simple yet precise description of the SOP. Like an implementation, the model can be executed to explore dynamic behaviors of the system, but unlike an implementation, the model omits low-level details that may get in the way of understanding the essential concepts.

The approach we take might be called “agile modeling” because of its similarities to agile programming. We work incrementally, assembling the model bit by bit. Our evolving model is at

every point something that can be executed. We formulate and run tests as we go, so that by the end we have not only the model itself but also a collection of *properties* that it satisfies.

To construct this model, we use *Alloy*, a language for modeling and analyzing software design. An Alloy model cannot be executed in the traditional sense of program execution. Instead, a model can be (1) *simulated* to produce an *instance*, which represents a valid scenario or configuration of a system, and (2) *checked* to see whether the model satisfies a desired property.

Despite the above similarities, agile modeling differs from agile programming in one key respect: Although we'll be running tests, we actually won't be writing any. Alloy's analyzer generates test cases automatically, and all that needs to be provided is the property to be checked. Needless to say, this saves a lot of trouble (and text). The analyzer actually executes all possible test cases up to a certain size (called a *scope*); this typically means generating all starting states with at most some number of objects, and then choosing operations and arguments to apply up to some number of steps. Because so many tests (typically billions) are executed, and because all possible configurations that a state can take are covered (albeit within the scope), this analysis tends to expose bugs more effectively than conventional testing (and is sometimes described as "bounded verification").

Simplifications

Because the SOP operates in the context of browsers, servers, HTTP, and so on, a complete description would be overwhelming. So our model (like all models) abstracts away irrelevant aspects, such as how network packets are structured and routed. But it also simplifies some relevant aspects, which means that the model cannot fully account for all possible security vulnerabilities.

For example, we treat HTTP requests like remote procedure calls, ignoring the fact that responses to requests might come out of order. We also assume that DNS (the domain name service) is static, so we cannot consider attacks in which a DNS binding changes during an interaction. In principle, though, it would be possible to extend our model to cover all these aspects, although it's in the very nature of security analysis that no model (even if it represents the entire codebase) can be guaranteed to be complete.

17.3 Roadmap

Here is the order in which we will proceed with our model of the SOP. We will begin by building models of three key components that we need in order for us to talk about the SOP: HTTP, the browser, and client-side scripting. We will build on top of these basic models to define what it means for a web application to be *secure*, and then introduce the SOP as a mechanism that attempts to achieve the required security properties.

We will then see that the SOP can sometimes be too restrictive, getting in the way of a web application's proper functioning. So we will introduce four different techniques that are commonly used to bypass the restrictions that are imposed by the policy.

Feel free to explore the sections in any order you'd like. If you are new to Alloy, we recommend starting with the first three sections (HTTP, Browser, and Script), as they introduce some of the basic concepts of the modeling language. While you are making your way through the chapter, we also encourage you to play with the models in the Alloy Analyzer; run them, explore the generated scenarios, and try making modifications and seeing their effects. It is freely available for download[1].

[1] `http://alloy.mit.edu`

17.4 Model of the Web

HTTP

The first step in building an Alloy model is to declare some sets of objects. Let's start with resources:

```
sig Resource {}
```

The keyword `sig` identifies this as an Alloy *signature* declaration. This introduces a set of resource objects; think of these, just like the objects of a class with no instance variables, as blobs that have identity but no content. When the analysis runs, this set will be determined, just as a class in an object-oriented language comes to denote a set of objects when the program executes.

Resources are named by URLs (*uniform resource locators*):

```
sig Url {
  protocol: Protocol,
  host: Domain,
  port: lone Port,
  path: Path
}
sig Protocol, Port, Path {}
sig Domain { subsumes: set Domain }
```

Here we have five signature declarations, introducing a set of URLs and four additional sets for each of the basic kinds of objects they comprise. Within the URL declaration, we have four *fields*. Fields are like instance variables in a class; if `u` is a URL, for example, then `u.protocol` would represent the protocol of that URL (just like dot in Java). But in fact, as we'll see later, these fields are relations. You can think of each one as if it were a two-column database table. Thus `protocol` is a table with the first column containing URLs and the second column containing protocols. And the innocuous looking dot operator is in fact a rather general kind of relational join, so that you could also write `protocol.p` for all the URLs with a protocol `p`—but more on that later.

Note that paths, unlike URLs, are treated as if they have no structure—a simplification. The keyword `lone` (which can be read "less than or equal to one") says that each URL has at most one port. The path is the string that follows the host name in the URL, and which (for a simple static server) corresponds to the file path of the resource; we're assuming that it's always present, but can be an empty path.

Let us introduce clients and servers, each of which contains a mapping from paths to resources:

```
abstract sig Endpoint {}
abstract sig Client extends Endpoint {}
abstract sig Server extends Endpoint {
  resources: Path -> lone Resource
}
```

The `extends` keyword introduces a subset, so the set `Client` of all clients, for example, is a subset of the set `Endpoint` of all endpoints. Extensions are disjoint, so no endpoint is both a client and a server. The `abstract` keyword says that all extensions of a signature exhaust it, so its occurrence in the declaration of `Endpoint`, for example, says that every endpoint must belong to one of the subsets (at this point, `Client` and `Server`). For a server `s`, the expression `s.resources` will denote a map from paths to resources (hence the arrow in the declaration). Recall that each field is actually

a relation that includes the owning signature as a first column, so this field represents a three-column relation on `Server`, `Path` and `Resource`.

To map a URL to a server, we introduce a set `Dns` of domain name servers, each with a mapping from domains to servers:

```
one sig Dns {
  map: Domain -> Server
}
```

The keyword `one` in the signature declaration means that (for simplicity) we're going to assume exactly one domain name server, and there will be a single DNS mapping, given by the expression `Dns.map`. Again, as with the serving resources, this could be dynamic (and in fact there are known security attacks that rely on changing DNS bindings during an interaction) but we're simplifying.

In order to model HTTP requests, we also need the concept of *cookies*, so let's declare them:

```
sig Cookie {
  domains: set Domain
}
```

Each cookie is scoped with a set of domains; this captures the fact that a cookie can apply to `*.mit.edu`, which would include all domains with the suffix `mit.edu`.

Finally, we can put this all together to construct a model of HTTP requests:

```
abstract sig HttpRequest extends Call {
  url: Url,
  sentCookies: set Cookie,
  body: lone Resource,
  receivedCookies: set Cookie,
  response: lone Resource,
}{
  from in Client
  to in Dns.map[url.host]
}
```

We're modeling an HTTP request and response in a single object; the `url`, `sentCookies` and `body` are sent by the client, and the `receivedCookies` and `response` are sent back by the server.

When writing the `HttpRequest` signature, we found that it contained generic features of calls, namely that they are from and to particular things. So we actually wrote a little Alloy module that declares the `Call` signature, and to use it here we need to import it:

```
open call[Endpoint]
```

It's a polymorphic module, so it's instantiated with `Endpoint`, the set of things calls are from and to. (The module appears in full in Section 17.10.)

Following the field declarations in `HttpRequest` is a collection of constraints. Each of these constraints applies to all members of the set of HTTP requests. The constraints say that (1) each request comes from a client, and (2) each request is sent to one of the servers specified by the URL host under the DNS mapping.

One of the prominent features of Alloy is that a model, no matter how simple or detailed, can be executed at any time to generate sample system instances. Let's use the `run` command to ask the Alloy Analyzer to execute the HTTP model that we have so far:

```
run {} for 3    -- generate an instance with up to 3 objects of every signature type
```

As soon as the analyzer finds a possible instance of the system, it automatically produces a diagram of the instance, like in Figure 17.1.

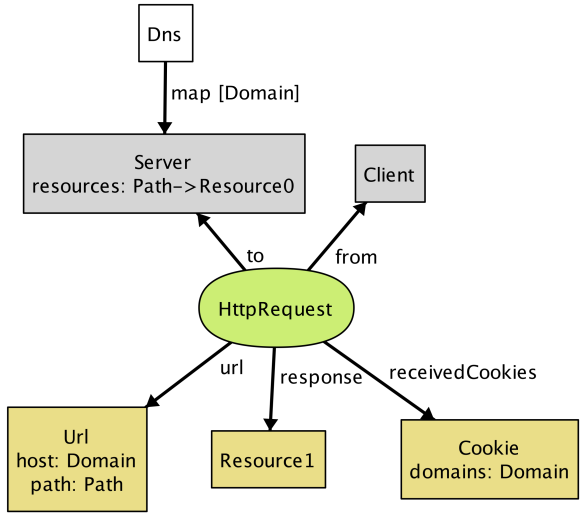

Figure 17.1: A possible instance

This instance shows a client (represented by node `Client`) sending an `HttpRequest` to `Server`, which, in response, returns a resource object and instructs the client to store `Cookie` at `Domain`.

Even though this is a tiny instance with seemingly few details, it signals a flaw in our model. Note that the resource returned from the request (`Resource1`) does not exist in the server. We neglected to specify an obvious fact about the server; namely, that every response to a request is a resource that the server stores. We can go back to our definition of `HttpRequest` and add a constraint:

```
abstract sig HttpRequest extends Call { ... }{
  ...
  response = to.resources[url.path]
}
```

Rerunning now produces instances without the flaw.

Instead of generating sample instances, we can ask the analyzer to *check* whether the model satisfies a property. For example, one property we might want is that when a client sends the same request multiple times, it always receives the same response back:

```
check {
    all r1, r2: HttpRequest | r1.url = r2.url implies r1.response = r2.response
} for 3
```

Given this `check` command, the analyzer explores every possible behavior of the system (up to the specified bound), and when it finds one that violates the property, displays that instance as a *counterexample*, as shown in Figure 17.2 and Figure 17.3.

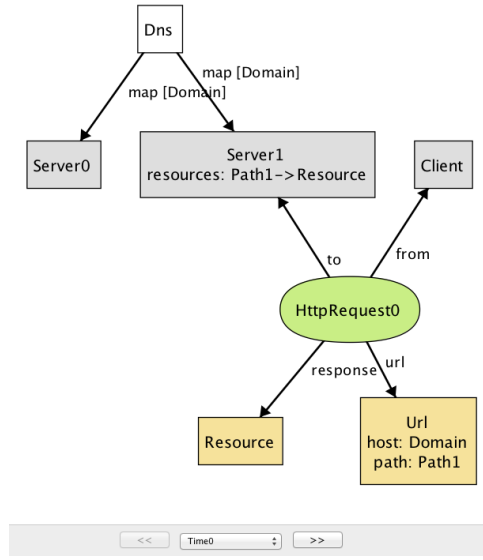

Figure 17.2: Counterexample at time 0

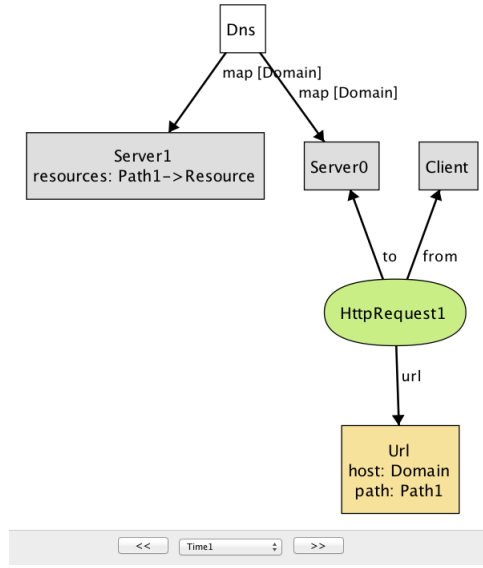

Figure 17.3: Counterexample at time 1

This counterexample again shows an HTTP request being made by a client, but with two different servers. (In the Alloy visualizer, objects of the same type are distinguished by appending numeric suffixes to their names; if there is only one object of a given type, no suffix is added. Every name that appears in a snapshot diagram is the name of an object. So—perhaps confusingly at first sight—the names `Domain`, `Path`, `Resource`, `Url` all refer to individual objects, not to types.)

Note that while the DNS maps `Domain` to both `Server0` and `Server1` (in reality, this is a common practice for load balancing), only `Server1` maps `Path` to a resource object, causing `HttpRequest1`

to result in an empty response: another error in our model. To fix this, we add an Alloy *fact* recording that any two servers to which DNS maps a single host provide the same set of resources:

```
fact ServerAssumption {
  all s1, s2: Server |
    (some Dns.map.s1 & Dns.map.s2) implies s1.resources = s2.resources
}
```

When we re-run the `check` command after adding this fact, the analyzer no longer reports any counterexamples for the property. This doesn't mean the property has been proven to be true, since there might be a counterexample in a larger scope. But it is unlikely that the property is false, since the analyzer has tested all possible instances involving 3 objects of each type.

If desired, however, we can re-run the analysis with a larger scope for increased confidence. For example, running the above check with the scope of 10 still does not produce any counterexample, suggesting that the property is likely to be valid. However, keep in mind that given a larger scope, the analyzer needs to test a greater number of instances, and so it will likely take longer to complete.

Browser

Let's now introduce browsers into our model:

```
sig Browser extends Client {
  documents: Document -> Time,
  cookies: Cookie -> Time,
}
```

This is our first example of a signature with *dynamic fields*. Alloy has no built-in notions of time or behavior, which means that a variety of idioms can be used. In this model, we're using a common idiom in which you introduce a notion of `Time`, and attach it as a final column for every time-varying field. For example, the expression `b.cookies.t` represents the set of cookies that are stored in browser `b` at a particular time `t`. Likewise, the `documents` field associates a set of documents with each browser at a given time. (For more details about how we model the dynamic behavior, see Section 17.10.)

Documents are created from a response to an HTTP request. They can also be destroyed if, for example, the user closes a tab or the browser, but we leave this out of the model. A document has a URL (the one from which the document was originated), some content (the DOM), and a domain:

```
sig Document {
  src: Url,
  content: Resource -> Time,
  domain: Domain -> Time
}
```

The inclusion of the `Time` column for the latter two fields tells us that they can vary over time, and its omission for the first (`src`, representing the source URL of the document) indicates that the source URL is fixed.

To model the effect of an HTTP request on a browser, we introduce a new signature, since not all HTTP requests will originate at the level of the browser; the rest will come from scripts.

```
sig BrowserHttpRequest extends HttpRequest {
  doc: Document
}{
  -- the request comes from a browser
  from in Browser
  -- the cookies being sent exist in the browser at the time of the request
  sentCookies in from.cookies.start
  -- every cookie sent must be scoped to the url of the request
  all c: sentCookies | url.host in c.domains

  -- a new document is created to display the content of the response
  documents.end = documents.start + from -> doc
  -- the new document has the response as its contents
  content.end = content.start ++ doc -> response
  -- the new document has the host of the url as its domain
  domain.end = domain.start ++ doc -> url.host
  -- the document's source field is the url of the request
  doc.src = url

  -- new cookies are stored by the browser
  cookies.end = cookies.start + from -> sentCookies
}
```

This kind of request has one new field, `doc`, representing the document created in the browser from the resource returned by the request. As with `HttpRequest`, the behavior is described as a collection of constraints. Some of these say when the call can happen: for example, that the call has to come from a browser. Some constrain the arguments of the call: for example, that the cookies must be scoped appropriately. And some constrain the effect, using a common idiom that relates the value of a relation after the call to its value before.

For example, to understand the constraint `documents.end = documents.start + from -> doc` remember that `documents` is a 3-column relation on browsers, documents and times. The fields `start` and `end` come from the declaration of `Call` (which we haven't seen, but is included in the listing at the end), and represent the times at the beginning and end of the call. The expression `documents.end` gives the mapping from browsers to documents when the call has ended. So this constraint says that after the call, the mapping is the same, except for a new entry in the table mapping `from` to `doc`.

Some constraints use the `++` relational *override* operator: `e1 ++ e2` contains all tuples of `e2`, and additionally, any tuples of `e1` whose first element is not the first element of a tuple in `e2`. For example, the constraint `content.end = content.start ++ doc -> response` says that after the call, the `content` mapping will be updated to map `doc` to `response` (overriding any previous mapping of `doc`). If we were to use the union operator `+` instead, then the same document might (incorrectly) be mapped to multiple resources in the after state.

Script

Next, we will build on the HTTP and browser models to introduce *client-side scripts*, which represent pieces of code (typically in JavaScript) executing inside a browser document (`context`).

```
sig Script extends Client { context: Document }
```

A script is a dynamic entity that can perform two different kinds of action: (1) it can make HTTP requests (i.e., Ajax requests) and (2) it can perform browser operations to manipulate the content and properties of a document. The flexibility of client-side scripts is one of the main catalysts of the rapid development of Web 2.0, but is also the reason why the SOP was created in the first place. Without the SOP, scripts would be able to send arbitrary requests to servers, or freely modify documents inside the browser—which would be bad news if one or more of the scripts turned out to be malicious.

A script can communicate to a server by sending an `XmlHttpRequest`:

```
sig XmlHttpRequest extends HttpRequest {}{
  from in Script
  noBrowserChange[start, end] and noDocumentChange[start, end]
}
```

An `XmlHttpRequest` can be used by a script to send/receive resources to/from a server, but unlike `BrowserHttpRequest`, it does not immediately result in the creation of a new page or other changes to the browser and its documents. To say that a call does not modify these aspects of the system, we define predicates `noBrowserChange` and `noDocumentChange`:

```
pred noBrowserChange[start, end: Time] {
  documents.end = documents.start and cookies.end = cookies.start
}
pred noDocumentChange[start, end: Time] {
  content.end = content.start and domain.end = domain.start
}
```

What kind of operations can a script perform on documents? First, we introduce a generic notion of *browser operations* to represent a set of browser API functions that can be invoked by a script:

```
abstract sig BrowserOp extends Call { doc: Document }{
  from in Script and to in Browser
  doc + from.context in to.documents.start
  noBrowserChange[start, end]
}
```

Field `doc` refers to the document that will be accessed or manipulated by this call. The second constraint in the signature facts says that both `doc` and the document in which the script executes (`from.context`) must be documents that currently exist inside the browser. Finally, a `BrowserOp` may modify the state of a document, but not the set of documents or cookies that are stored in the browser. (Actually, cookies can be associated with a document and modified using a browser API, but we omit this detail for now.)

A script can read from and write to various parts of a document (usually called DOM elements). In a typical browser, there are a large number of API functions for accessing the DOM (e.g., `document.getElementById`), but enumerating all of them is not important for our purpose. Instead, we will simply group them into two kinds—`ReadDom` and `WriteDom`—and model modifications as wholesale replacements of the entire document:

```
sig ReadDom extends BrowserOp { result: Resource }{
  result = doc.content.start
  noDocumentChange[start, end]
}
```

```
sig WriteDom extends BrowserOp { newDom: Resource }{
  content.end = content.start ++ doc -> newDom
  domain.end = domain.start
}
```

`ReadDom` returns the content of the target document, but does not modify it; `WriteDom`, on the other hand, sets the new content of the target document to `newDom`.

In addition, a script can modify various properties of a document, such as its width, height, domain, and title. For our discussion of the SOP, we are only interested in the domain property, which we will introduce in a later section.

17.5 Example Applications

As we've seen earlier, given a `run` or `check` command, the Alloy Analyzer generates a scenario (if it exists) that is consistent with the description of the system in the model. By default, the analyzer arbitrarily picks *any* one of the possible system scenarios (up to the specified bound), and assigns numeric identifiers to signature instances (`Server0`, `Browser1`, etc.) in the scenario.

Sometimes, we may wish to analyze the behavior of a *particular* web application, instead of exploring scenarios with a random configuration of servers and clients. For example, imagine that we wish to build an email application that runs inside a browser (like Gmail). In addition to providing basic email features, our application might display a banner from a third-party advertisement service, which is controlled by a potentially malicious actor.

In Alloy, the keywords `one sig` introduce a *singleton* signature containing exactly one object; we saw an example above with `Dns`. This syntax can be used to specify concrete atoms. For example, to say that there is one inbox page and one ad banner (each of which is a document) we can write:

```
one sig InboxPage, AdBanner extends Document {}
```

With this declaration, every scenario that Alloy generates will contain at least these two `Document` objects.

Likewise, we can specify particular servers, domains and so on, with a constraint (which we've called `Configuration`) to specify the relationships between them:

```
one sig EmailServer, EvilServer extends Server {}
one sig EvilScript extends Script {}
one sig EmailDomain, EvilDomain extends Domain {}
fact Configuration {
  EvilScript.context = AdBanner
  InboxPage.domain.first = EmailDomain
  AdBanner.domain.first = EvilDomain
  Dns.map = EmailDomain -> EmailServer + EvilDomain -> EvilServer
}
```

For example, the last constraint in the fact specifies how the DNS is configured to map domain names for the two servers in our system. Without this constraint, the Alloy Analyzer may generate scenarios where `EmailDomain` is mapped to `EvilServer`, which are not of interest to us. (In practice, such a mapping may be possible due to an attack called *DNS spoofing*, but we will rule it out from our model since it lies outside the class of attacks that the SOP is designed to prevent.)

Let us introduce two additional applications: an online calendar and a blog site:

```
one sig CalendarServer, BlogServer extends Document {}
one sig CalendarDomain, BlogDomain extends Domain {}
```

We should update the constraint about the DNS mapping above to incorporate the domain names for these two servers:

```
fact Configuration {
  ...
  Dns.map = EmailDomain -> EmailServer + EvilDomain -> EvilServer +
            CalendarDomain -> CalendarServer + BlogDomain -> BlogServer
}
```

In addition, let us say that that the email, blog, and calendar applications are all developed by a single organization, and thus, share the same base domain name. Conceptually, we can think of `EmailServer` and `CalendarServer` having subdomains `email` and `calendar`, sharing `example.com` as the common superdomain. In our model, this can be represented by introducing a domain name that *subsumes* others:

```
one sig ExampleDomain extends Domain {}{
  subsumes = EmailDomain + EvilDomain + CalendarDomain + this
}
```

Note that `this` is included as a member of `subsumes`, since every domain name subsumes itself.

There are other details about these applications that we omit here (see `example.als` for the full model). But we will revisit these applications as our running example throughout the remainder of this chapter.

17.6 Security Properties

Before we get to the SOP itself, there is an important question that we have not discussed yet: What exactly do we mean when we say our system is *secure*?

Not surprisingly, this is a tricky question to answer. For our purposes, we will turn to two well-studied concepts in information security—*confidentiality* and *integrity*. Both of these concepts talk about how information should be allowed to pass through the various parts of the system. Roughly, *confidentiality* means that a critical piece of data should only be accessible to parts that are deemed trusted, and *integrity* means that trusted parts only rely on data that have not been maliciously tampered with.

Dataflow Properties

In order to specify these security properties more precisely, we first need to define what it means for a piece of data to *flow* from one part of the system to another. In our model so far, we have described interactions between two endpoints as being carried out through *calls*; e.g., a browser interacts with a server by making HTTP requests, and a script interacts with the browser by invoking browser API operations. Intuitively, during each call, a piece of data may flow from one endpoint to another as an *argument* or *return value* of the call. To represent this, we introduce a notion of `DataflowCall` into the model, and associate each call with a set of `args` and `returns` data fields:

```
sig Data in Resource + Cookie {}

sig DataflowCall in Call {
  args, returns: set Data,  --- arguments and return data of this call
}{
 this in HttpRequest implies
    args = this.sentCookies + this.body and
    returns = this.receivedCookies + this.response
 ...
}
```

For example, during each call of type `HttpRequest`, the client transfers `sentCookies` and `body` to the server, and receives `receivedCookies` and `response` as return values.

More generally, arguments flow from the sender of the call to the receiver, and return values flow from the receiver to the sender. This means that the only way for an endpoint to access a new piece of data is by receiving it as an argument of a call that the endpoint accepts, or a return value of a call that the endpoint invokes. We introduce a notion of `DataflowModule`, and assign field `accesses` to represent the set of data elements that the module can access at each time step:

```
sig DataflowModule in Endpoint {
  -- Set of data that this component initially owns
  accesses: Data -> Time
}{
  all d: Data, t: Time - first |
     -- This endpoint can only access a piece of data "d" at time "t" only when
    d -> t in accesses implies
      -- (1) It already had access in the previous time step, or
      d -> t.prev in accesses or
      -- there is some call "c" that ended at "t" such that
      some c: Call & end.t |
        -- (2) the endpoint receives "c" that carries "d" as one of its arguments or
        c.to = this and d in c.args or
        -- (3) the endpoint sends "c" that returns "d"
        c.from = this and d in c.returns
}
```

We also need to restrict data elements that a module can provide as arguments or return values of a call. Otherwise, we may get weird scenarios where a module can make a call with an argument that it has no access to.

```
sig DataflowCall in Call { ... } {
  -- (1) Any arguments must be accessible to the sender
  args in from.accesses.start
  -- (2) Any data returned from this call must be accessible to the receiver
  returns in to.accesses.start
}
```

Now that we have means to describe data flow between different parts of the system, we are (almost) ready to state security properties that we care about. But recall that confidentiality and integrity are *context-dependent* notions; these properties make sense only if we can talk about some agents within the system as being trusted (or malicious). Similarly, not all information is equally

important: we need to distinguish between data elements that we consider to be critical or malicious (or neither):

```
sig TrustedModule, MaliciousModule in DataflowModule {}
sig CriticalData, MaliciousData in Data {}
```

Then, the confidentiality property can be stated as an *assertion* on the flow of critical data into non-trusted parts of the system:

```
// No malicious module should be able to access critical data
assert Confidentiality {
  no m: Module - TrustedModule, t: Time |
    some CriticalData & m.accesses.t
}
```

The integrity property is the dual of confidentiality:

```
// No malicious data should ever flow into a trusted module
assert Integrity {
  no m: TrustedModule, t: Time |
    some MaliciousData & m.accesses.t
}
```

Threat Model

A threat model describes a set of actions that an attacker may perform in an attempt to compromise a security property of a system. Building a threat model is an important step in any secure system design; it allows us to identify (possibly invalid) assumptions that we have about the system and its environment, and prioritize different types of risks that need to be mitigated.

In our model, we consider an attacker that can act as a server, a script or a client. As a server, the attacker may set up malicious web pages to solicit visits from unsuspecting users, who, in turn, may inadvertently send sensitive information to the attacker as part of a HTTP request. The attacker may create a malicious script that invokes DOM operations to read data from other pages and relays those data to the attacker's server. Finally, as a client, the attacker may impersonate a normal user and send malicious requests to a server in an attempt to access the user's data. We do not consider attackers that eavesdrop on the connection between different network endpoints; although it is a threat in practice, the SOP is not designed to prevent it, and thus it lies outside the scope of our model.

Checking Properties

Now that we have defined the security properties and the attacker's behavior, let us show how the Alloy Analyzer can be used to automatically check that those properties hold even in the presence of the attacker. When prompted with a `check` command, the analyzer explores *all* possible dataflow traces in the system and produces a counterexample (if one exists) that demonstrates how an assertion might be violated:

```
check Confidentiality for 5
```

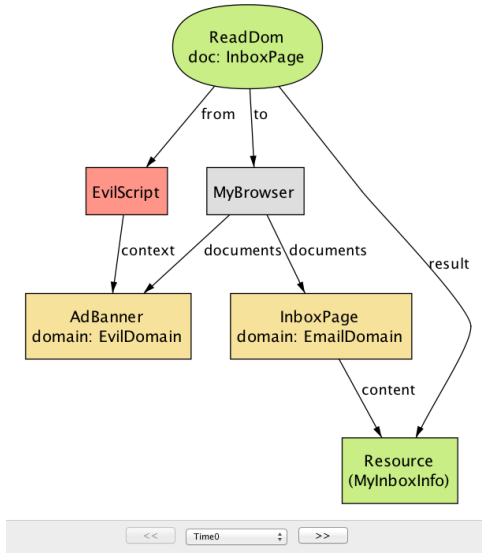

Figure 17.4: Confidentiality counterexample at time 0

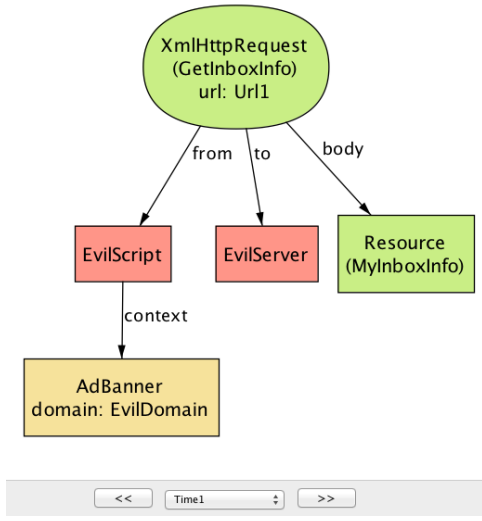

Figure 17.5: Confidentiality counterexample at time 1

For example, when checking the model of our example application against the confidentiality property, the analyzer generates the scenario seen in Figure 17.4 and Figure 17.5, which shows how `EvilScript` may access a piece of critical data (`MyInboxInfo`).

This counterexample involves two steps. In the first step (Figure 17.4), `EvilScript`, executing inside `AdBanner` from `EvilDomain`, reads the content of `InboxPage`, which originates from `EmailDomain`. In the next step (Figure 17.5), `EvilScript` sends the same content (`MyInboxInfo`) to `EvilServer` by making an `XmlHtttpRequest` call. The core of the problem here is that a script executing under one domain is able to read the content of a document from another domain; as we will see in the next section, this is exactly one of the scenarios that the SOP is designed to prevent.

There may be multiple counterexamples to a single assertion. Consider Figure 17.6, which shows a different way in which the system may violate the confidentiality property.

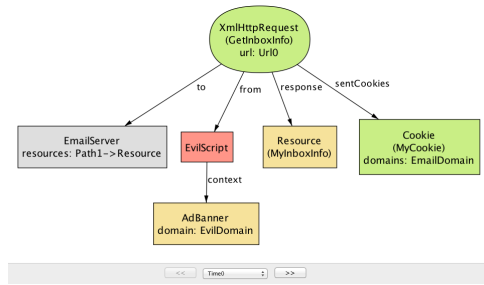

Figure 17.6: Another confidentiality violation

In this scenario, instead of reading the content of the inbox page, `EvilScript` directly makes a `GetInboxInfo` request to `EmailServer`. Note that the request includes a cookie (`MyCookie`), which is scoped to the same domain as the destination server. This is potentially dangerous, because if the cookie is used to represent the user's identity (e.g., a session cookie), `EvilScript` can effectively pretend to be the user and trick the server into responding with the user's private data (`MyInboxInfo`). Here, the problem is again related to the liberal ways in which a script may be used to access information across different domains—namely, that a script executing under one domain is able to make an HTTP request to a server with a different domain.

These two counterexamples tell us that extra measures are needed to restrict the behavior of scripts, especially since some of those scripts could be malicious. This is exactly where the SOP comes in.

17.7 Same-Origin Policy

Before we can state the SOP, the first thing we should do is to introduce the notion of an *origin*, which is composed of a protocol, host, and optional port:

```
sig Origin {
  protocol: Protocol,
  host: Domain,
  port: lone Port
}
```

For convenience, let us define a function that, given a URL, returns the corresponding origin:

```
fun origin[u: Url] : Origin {
    {o: Origin | o.protocol = u.protocol and o.host = u.host and o.port = u.port }
}
```

The SOP itself has two parts, restricting the ability of a script to (1) make DOM API calls and (2) send HTTP requests. The first part of the policy states that a script can only read from and write to a document that comes from the same origin as the script:

```
fact domSop {
  all o: ReadDom + WriteDom |  let target = o.doc, caller = o.from.context |
```

```
    origin[target] = origin[caller]
}
```

An instance such as the first script scenario (from the previous section) is not possible under `domSop`, since `Script` is not allowed to invoke `ReadDom` on a document from a different origin.

The second part of the policy says that a script cannot send an HTTP request to a server unless its context has the same origin as the target URL—effectively preventing instances such as the second script scenario.

```
fact xmlHttpReqSop {
  all x: XmlHttpRequest | origin[x.url] = origin[x.from.context.src]
}
```

As we can see, the SOP is designed to prevent the two types of vulnerabilities that could arise from actions of a malicious script; without it, the web would be a much more dangerous place than it is today.

It turns out, however, that the SOP can be *too* restrictive. For example, sometimes you *do* want to allow communication between two documents of different origins. By the above definition of an origin, a script from `foo.example.com` would not be able to read the content of `bar.example.com`, or send a HTTP request to `www.example.com`, because these are all considered distinct hosts.

In order to allow some form of cross-origin communication when necessary, browsers implemented a variety of mechanisms for relaxing the SOP. Some of these are more well-thought-out than others, and some have pitfalls that, when badly used, can undermine the security benefits of the SOP. In the following sections, we will describe the most common of these mechanisms, and discuss their potential security pitfalls.

17.8 Techniques for Bypassing the SOP

The SOP is a classic example of the tension between functionality and security; we want to make sure our sites are robust and functional, but the mechanism for securing it can sometimes get in the way. Indeed, when the SOP was initially introduced, developers ran into trouble building sites that made legitimate uses of cross-domain communication (e.g., mashups).

In this section, we will discuss four techniques that have been devised and frequently used by web developers to bypass the restrictions imposed by the SOP: (1) The `document.domain` property relaxation; (2) JSONP; (3) PostMessage; and (4) CORS. These are valuable tools, but if used without caution, may render a web application vulnerable to exactly the kinds of attacks that the SOP was designed to thwart in the first place.

Each of these four techniques is surprisingly complex, and if described in full detail, would merit its own chapter. So here we just give a brief impression of how they work, potential security problems that they introduce, and how to prevent these problems. In particular, we will ask the Alloy Analyzer to check, for each technique, whether it could be abused by an attacker to undermine the two security properties that we defined earlier:

```
check Confidentiality for 5
check Integrity for 5
```

Based on insights from the counterexamples that the analyzer generates, we will discuss guidelines for safely using these techniques without falling into security pitfalls.

Domain Property

As the first technique on our list, we will look at the use of the `document.domain` property as a way of bypassing the SOP. The idea behind this technique is to allow two documents from different origins to access each other's DOM simply by setting the `document.domain` property to the same value. So, for example, a script from `email.example.com` could read or write the DOM of a document from `calendar.example.com` if the scripts in both documents set the `document.domain` property to `example.com` (assuming both source URLs have also the same protocol and port).

We model the behavior of setting the `document.domain` property as a type of browser operation called `SetDomain`:

```
// Modify the document.domain property
sig SetDomain extends BrowserOp { newDomain: Domain }{
  doc = from.context
  domain.end = domain.start ++ doc -> newDomain
  -- no change to the content of the document
  content.end = content.start
}
```

The `newDomain` field represents the value to which the property should be set. There's a caveat, though: scripts can only set the domain property to a right-hand, fully qualified fragment of its hostname. (I.e., `email.example.com` can set it to `example.com` but not to `google.com`.) We use a fact to capture this rule about subdomains:

```
// Scripts can only set the domain property to only one that is a right-hand,
// fully-qualified fragment of its hostname
fact setDomainRule {
  all d: Document | d.src.host in (d.domain.Time).subsumes
}
```

If it weren't for this rule, any site could set the `document.domain` property to any value, which means that, for example, a malicious site could set the domain property to your bank domain, load your bank account in an iframe, and (assuming the bank page has set its domain property) read the DOM of your bank page.

Let us go back to our original definition of the SOP, and relax its restriction on DOM access in order to take into account the effect of the `document.domain` property. If two scripts set the property to the same value, and they have the same protocol and port, then these two scripts can interact with each other (that is, read and write each other's DOM).

```
fact domSop {
  -- For every successful read/write DOM operation,
  all o: ReadDom + WriteDom |  let target = o.doc, caller = o.from.context |
    -- (1) target and caller documents are from the same origin, or
    origin[target] = origin[caller] or
    -- (2) domain properties of both documents have been modified
    (target + caller in (o.prevs <: SetDomain).doc and
      -- ...and they have matching origin values.
      currOrigin[target, o.start] = currOrigin[caller, o.start])
}
```

Here, `currOrigin[d, t]` is a function that returns the origin of document `d` with the property `document.domain` at time `t` as its hostname.

It is worth pointing out that the `document.domain` properties for *both* documents must be *explictly* set sometime after they are loaded into the browser. Let us say that document A is loaded from `example.com`, and document B from `calendar.example.com` has its domain property modified to `example.com`. Even though the two documents now have the same domain property, they will *not* be able to interact with each other, unless document A also explictly sets its property to `example.com`. At first, this seems like a rather strange behavior. However, without this, various bad things can happen. For example, a site could be subject to a cross-site scripting attack from its subdomains: A malicious script in document B might modify its domain property to `example.com` and manipulate the DOM of document A, even though the latter never intended to interact with document B.

Analysis: Now that we have relaxed the SOP to allow cross-origin communication under certain circumstances, do the security guarantees of the SOP still hold? Let us ask the Alloy Analyzer to tell us whether the `document.domain` property could be abused by an attacker to access or tamper with a user's sensitive data.

Indeed, given the new, relaxed definition of the SOP, the analyzer generates a counterexample scenario to the confidentiality property:

```
check Confidentiality for 5
```

This scenario consists of five steps; the first three steps show a typical use of `document.domain`, where two documents from distinct origins, `CalendarPage` and `InboxPage`, communicate by setting their domain properties to a common value (`ExampleDomain`). The last two steps introduce another document, `BlogPage`, that has been compromised with a malicious script that attempts to access the content of the other two documents.

At the beginning of the scenario (Figure 17.7 and Figure 17.8), `InboxPage` and `CalendarPage` have domain properties with two distinct values (`EmailDomain` and `ExampleDomain`, respectively), so the browser will prevent them from accessing each other's DOM. The scripts running inside the documents (`InboxScript` and `CalendarScript`) each execute the `SetDomain` operation to modify their domain properties to `ExampleDomain` (which is allowed because `ExampleDomain` is a superdomain of the original domain).

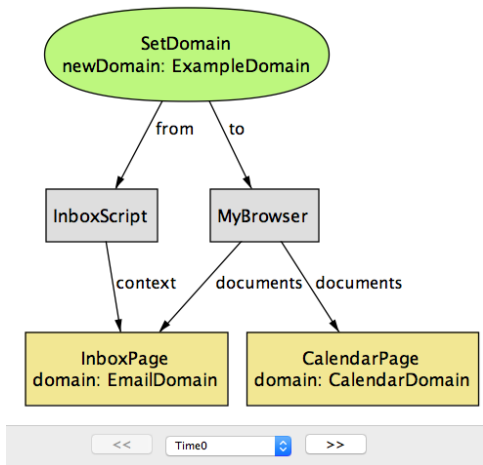

Figure 17.7: Cross-origin counterexample at time 0

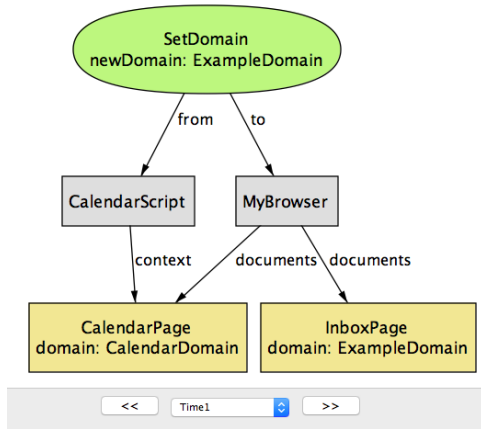

Figure 17.8: Cross-origin counterexample at time 1

Having done this, they can now access each other's DOM by executing `ReadDom` or `WriteDom` operations, as in Figure 17.9.

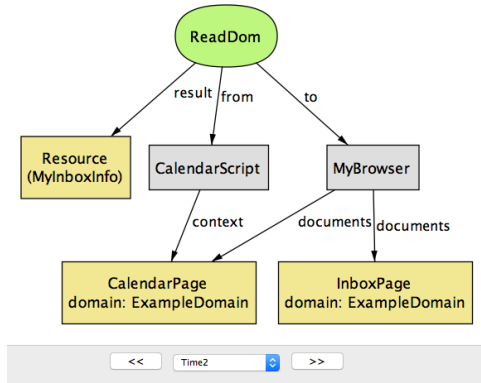

Figure 17.9: Cross-origin counterexample at time 2

Note that when you set the domain of `email.example.com` and `calendar.example.com` to `example.com`, you are allowing not only these two pages to communicate between each other, but also *any* other page that has `example.com` as a superdomain (e.g., `blog.example.com`). An attacker also realizes this, and constructs a special script (`EvilScript`) that runs inside the attacker's blog page (`BlogPage`). In the next step (Figure 17.10), the script executes the `SetDomain` operation to modify the domain property of `BlogPage` to `ExampleDomain`.

Now that `BlogPage` has the same domain property as the other two documents, it can successfully execute the `ReadDOM` operation to access their content (Figure 17.11.)

This attack points out one crucial weakness of the domain property method for cross-origin communication: The security of an application that uses this method is only as strong as the weakest link in all of the pages that share the same base domain. We will shortly discuss another method called PostMessage, which can be used for a more general class of cross-origin communication while also being more secure.

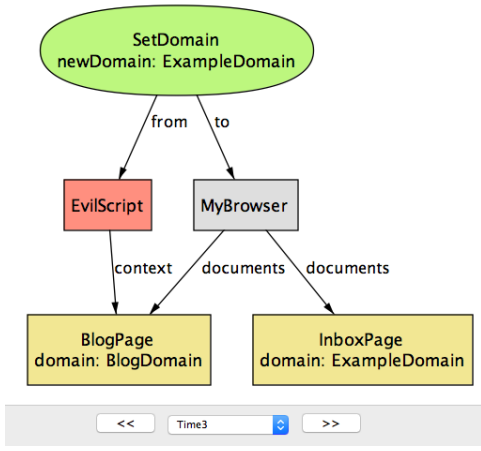

Figure 17.10: Cross-origin counterexample at time 3

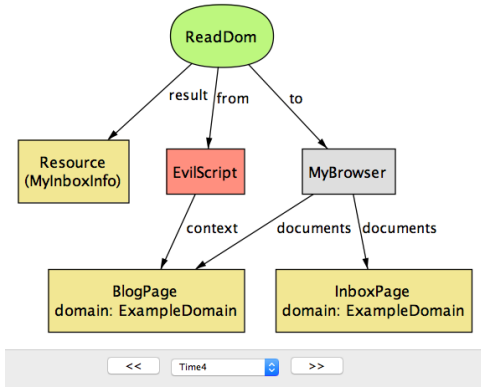

Figure 17.11: Cross-origin counterexample at time 4

JSON with Padding (JSONP)

Before the introduction of CORS (which we will discuss shortly), JSONP was perhaps the most popular technique for bypassing the SOP restriction on XMLHttpRequest, and still remains widely used today. JSONP takes advantage of the fact that script inclusion tags in HTML (i.e., `<script>`) are exempt from the SOP*; that is, you can include a script from *any* URL, and the browser readily executes it in the current document:

(* Without this exemption, it would not be possible to load JavaScript libraries, such as JQuery, from other domains.)

```
<script src="http://www.example.com/myscript.js"></script>
```

A script tag can be used to obtain code, but how do we use it to receive arbitrary *data* (e.g., a JSON object) from a different domain? The problem is that the browser expects the content of `src` to be a piece of JavaScript code, and so simply having it point at a data source (e.g., a JSON or HTML file) results in a syntax error.

One workaround is to wrap the desired data inside a string that the browser recognizes as valid JavaScript code; this string is sometimes called *padding* (hence the name "JSON with padding").

This padding could be any arbitrary JavaScript code, but conventionally, it is the name of a callback function (already defined in the current document) that is to be executed on the response data:

```
<script src="http://www.example.com/mydata?jsonp=processData"></script>
```

The server on `www.example.com` recognizes it as a JSONP request, and wraps the requested data inside the `jsonp` parameter:

```
processData(mydata)
```

which is a valid JavaScript statement (namely, the application of function "processData" on value "mydata"), and is executed by the browser in the current document.

In our model, JSONP is modeled as a kind of HTTP request that includes the identifier of a callback function in the field `padding`. After receiving a JSONP request, the server returns a response that has the requested resource (`payload`) wrapped inside the callback function (`cb`).

```
sig CallbackID {}  // identifier of a callback function
// Request sent as a result of <script> tag
sig JsonpRequest in BrowserHttpRequest {
  padding: CallbackID
}{
  response in JsonpResponse
}
sig JsonpResponse in Resource {
  cb: CallbackID,
  payload: Resource
}
```

When the browser receives the response, it executes the callback function on the payload:

```
sig JsonpCallback extends EventHandler {
  cb: CallbackID,
  payload: Resource
}{
  causedBy in JsonpRequest
  let resp = causedBy.response |
    cb = resp.@cb and
    -- result of JSONP request is passed on as an argument to the callback
    payload = resp.@payload
}
```

(`EventHandler` is a special type of call that must take place sometime after another call, which is denoted by `causedBy`; we will use event handlers to model actions that are performed by scripts in response to browser events.)

Note that the callback function executed is the same as the one that's included in the response (`cb = resp.@cb`), but *not* necessarily the same as `padding` in the original JSONP request. In other words, for the JSONP communication to work, the server is responsible for properly constructing a response that includes the original padding as the callback function (i.e., ensure that `JsonRequest.padding = JsonpResponse.cb`). In principle, the server can choose to include any callback function (or any piece of JavaScript), including one that has nothing to do with `padding` in the request. This

highlights a potential risk of JSONP: the server that accepts the JSONP requests must be trustworthy and secure, because it has the ability to execute any piece of JavaScript code in the client document.

Analysis: Checking the `Confidentiality` property with the Alloy Analyzer returns a counterexample that shows one potential security risk of JSONP. In this scenario, the calendar application (`CalendarServer`) makes its resources available to third-party sites using a JSONP endpoint (`GetSchedule`). To restrict access to the resources, `CalendarServer` only sends back a response with the schedule for a user if the request contains a cookie that correctly identifies that user.

Note that once a server provides an HTTP endpoint as a JSONP service, anyone can make a JSONP request to it, including malicious sites. In this scenario, the ad banner page from `EvilServer` includes a *script* tag that causes a `GetSchedule` request, with a callback function called `Leak` as `padding`. Typically, the developer of `AdBanner` does not have direct access to the victim user's session cookie (`MyCookie`) for `CalendarServer`. However, because the JSONP request is being sent to `CalendarServer`, the browser automatically includes `MyCookie` as part of the request; `CalendarServer`, having received a JSONP request with `MyCookie`, will return the victim's resource (`MySchedule`) wrapped inside the padding `Leak` (Figure 17.12.)

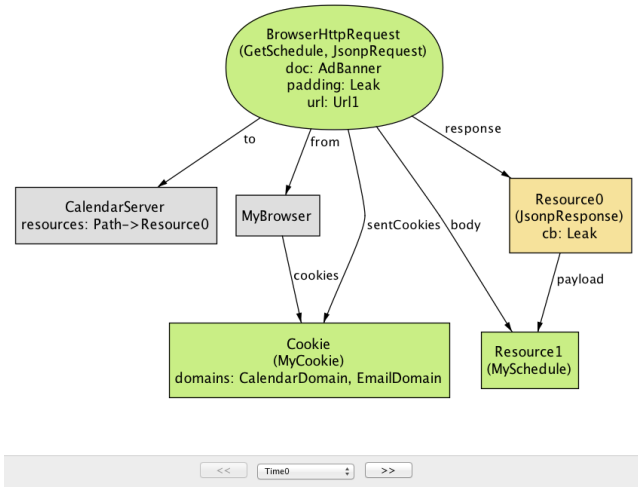

Figure 17.12: JSONP counterexample at time 0

In the next step, the browser interprets the JSONP response as a call to `Leak(MySchedule)` (Figure 17.13). The rest of the attack is simple; `Leak` can simply be programmed to forward the input argument to `EvilServer`, allowing the attacker to access the victim's sensitive information.

This attack, an example of *cross-site request forgery* (CSRF), shows an inherent weakness of JSOPN; *any* site on the web can make a JSONP request simply by including a `<script>` tag and access the payload inside the padding. The risk can be mitigated in two ways: (1) ensure that a JSONP request never returns sensitive data, or (2) use another mechanism in place of cookies (e.g., secret tokens) to authorize the request.

PostMessage

PostMessage is a new feature in HTML5 that allows scripts from two documents (of possibly different origins) to communicate with each other. It offers a more disciplined alternative to the method of setting the `domain` property, but brings its own security risks.

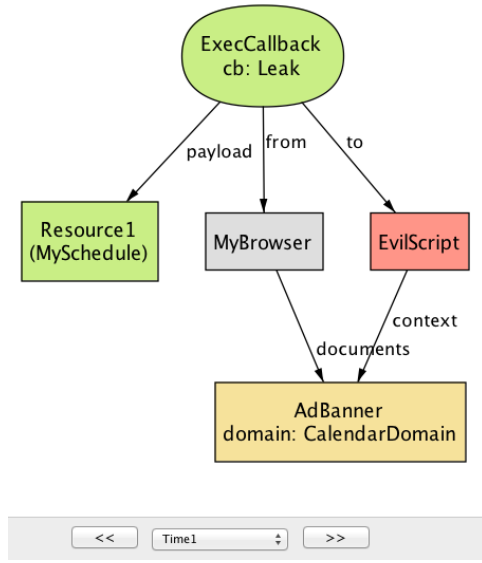

Figure 17.13: JSONP counterexample at time 1

`PostMessage` is a browser API function that takes two arguments: (1) the data to be sent (`message`), and (2) the origin of the document receiving the message (`targetOrigin`):

```
sig PostMessage extends BrowserOp {
  message: Resource,
  targetOrigin: Origin
}
```

To receive a message from another document, the receiving document registers an event handler that is invoked by the browser as a consequence of a `PostMessage`:

```
sig ReceiveMessage extends EventHandler {
  data: Resource,
  srcOrigin: Origin
}{
  causedBy in PostMessage
  -- "ReceiveMessage" event is sent to the script with the correct context
  origin[to.context.src] = causedBy.targetOrigin
  -- messages match
  data = causedBy.@message
  -- the origin of the sender script is provided as "srcOrigin" param
  srcOrigin = origin[causedBy.@from.context.src]
}
```

The browser passes two parameters to `ReceiveMessage`: a resource (`data`) that corresponds to the message being sent, and the origin of the sender document (`srcOrigin`). The signature fact contains four constraints to ensure that each `ReceiveMessage` is well-formed with respect to its corresponding `PostMessage`.

Analysis: Again, let us ask the Alloy Analyzer whether `PostMessage` is a secure way of performing cross-origin communication. This time, the analyzer returns a counterexample for the `Integrity`

property, meaning the attacker is able to exploit a weakness in `PostMessage` to introduce malicious data into a trusted application.

Note that by default, the PostMessage mechanism does not restrict who is allowed to send PostMessage; in other words, any document can send a message to another document as long as the latter has registered a `ReceiveMessage` handler. For example, in the following instance generated from Alloy, `EvilScript`, running inside `AdBanner`, sends a malicious `PostMessage` to a document with the target origin of `EmailDomain` (Figure 17.14.)

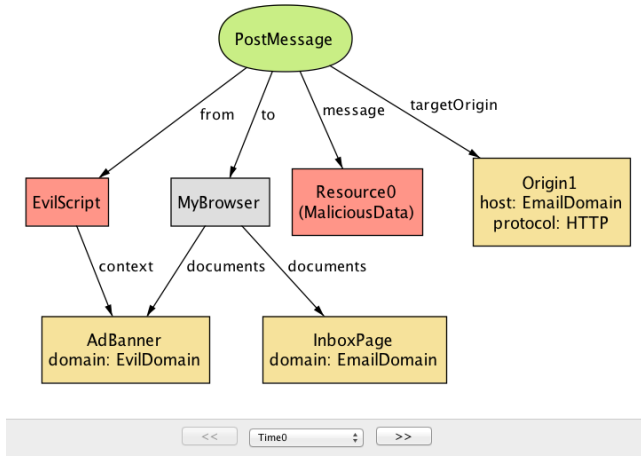

Figure 17.14: PostMessage counterexample at time 0

The browser then forwards this message to the document(s) with the corresponding origin (in this case, `InboxPage`). Unless `InboxScript` specifically checks the value of `srcOrigin` to filter out messages from unwanted origins, `InboxPage` will accept the malicious data, possibly leading to further security attacks. (For example, it may embed a piece of JavaScript to carry out an XSS attack.) This is shown in Figure 17.14.

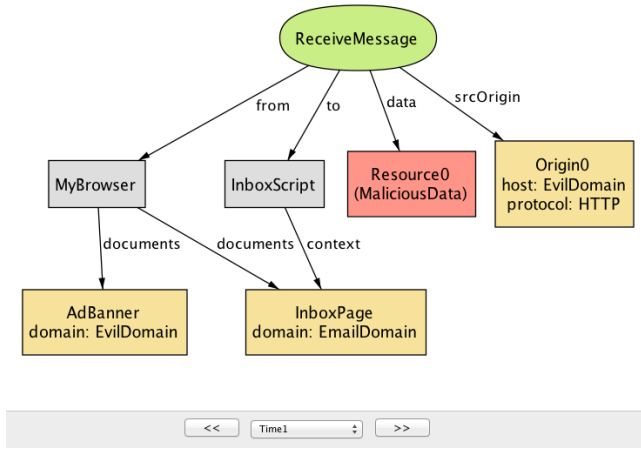

Figure 17.15: PostMessage counterexample at time 1

As this example illustrates, `PostMessage` is not secure by default, and it is the responsibility of

the receiving document to *additionally* check the `srcOrigin` parameter to ensure that the message is coming from a trustworthy document. Unfortunately, in practice, many sites omit this check, enabling a malicious document to inject bad content as part of a `PostMessage`[2].

However, the omission of the origin check may not simply be the result of programmer ignorance. Implementing an appropriate check on an incoming PostMessage can be tricky; in some applications, it is hard to determine in advance the list of trusted origins from which messages are expected to be received. (In some apps, this list may even change dynamically.) This, again, highlights the tension between security and functionality: PostMessage can be used for secure cross-origin communication, but only when a whitelist of trusted origins is known.

Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a mechanism designed to allow a server to share its resources with sites from different origins. In particular, CORS can be used by a script from one origin to make requests to a server with a different origin, effectively bypassing the restriction of the SOP on cross-origin Ajax requests.

Briefly, a typical CORS process involves two steps: (1) a script wanting to access a resource from a foreign server includes, in its request, an "Origin" header that specifies the origin of the script, and (2) the server includes an "Access-Control-Allow-Origin" header as part of its response, indicating a set of origins that are allowed to access the server's resource. Normally, without CORS, a browser would prevent the script from making a cross-origin request in the first place, conforming to the SOP. However, with CORS enabled, the browser allows the script to send the request and access its response, but *only if* "Origin" is one of the origins specified in "Access-Control-Allow-Origin".

(CORS additionally includes a notion of *preflight* requests, not discussed here, to support complex types of cross-origin requests besides GETs and POSTs.)

In Alloy, we model a CORS request as a special kind of `XmlHttpRequest`, with two extra fields `origin` and `allowedOrigins`:

```
sig CorsRequest in XmlHttpRequest {
  -- "origin" header in request from client
  origin: Origin,
  -- "access-control-allow-origin" header in response from server
  allowedOrigins: set Origin
}{
  from in Script
}
```

We then use an Alloy fact `corsRule` to describe what constitutes a valid CORS request:

```
fact corsRule {
  all r: CorsRequest |
    -- the origin header of a CORS request matches the script context
    r.origin = origin[r.from.context.src] and
    -- the specified origin is one of the allowed origins
    r.origin in r.allowedOrigins
}
```

[2]Sooel Son and Vitaly Shmatikov. *The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites*. Network and Distributed System Security Symposium (NDSS), 2013.

Analysis: Can CORS be misused in a way that would allow the attacker to compromise the security of a trusted site? When prompted, the Alloy Analyzer returns a simple counterexample for the `Confidentiality` property.

Here, the developer of the calendar application decides to share some of its resources with other applications by using the CORS mechanism. Unfortunately, `CalendarServer` is configured to return `Origin` (which represents the set of all origin values) for the `access-control-allow-origin` header in CORS responses. As a result, a script from any origin, including `EvilDomain`, is allowed to make a cross-site request to `CalendarServer` and read its response (Figure 17.16).

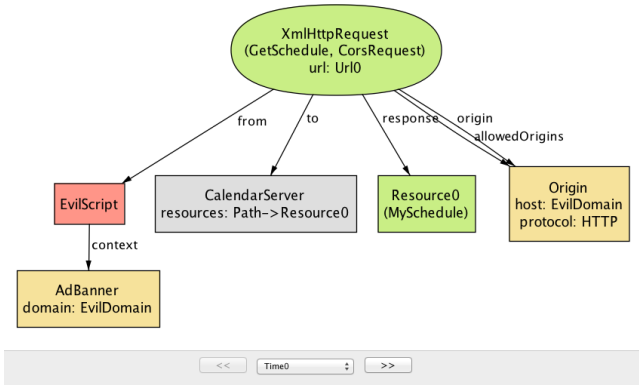

Figure 17.16: CORS counterexample

This example highlights one common mistake that developers make with CORS: Using the wildcard value "*" as the value of "access-control-allow-origin" header, allowing any site to access a resource on the server. This access pattern is appropriate if the resource is considered public and accessible to anyone. However, it turns out that many sites use "*" as the default value even for private resources, inadvertently allowing malicious scripts to access them through CORS requests[3].

Why would a developer ever use the wildcard? It turns out that specifying the allowed origins can be tricky, since it may not be clear at design time which origins should be granted access at runtime (similar to the PostMessage issue discusssed above). A service may, for example, allow third-party applications to subscribe dynamically to its resources.

17.9 Conclusion

In this chapter, we set out to construct a document that provides a clear understanding of the SOP and its related mechanisms by building a *model* of the policy in a language called Alloy. Our model of the SOP is not an implementation in the traditional sense, and can't be deployed for use, unlike artifacts shown in other chapters. Instead, we wanted to demonstrate the key elements behind our approach to "agile modeling": (1) starting out with a small, abstract model of the system and *incrementally* adding details as necessary, (2) specifying *properties* that the system is expected to satisfy, and (3) applying *rigorous analysis* to explore potential flaws in the design of the system. Of course, this chapter was written long after the SOP was first introduced, but we believe that this type of modeling would potentially be even more beneficial if it is done during the early stage of system design.

[3] Sebastian Lekies, Martin Johns, and Walter Tighzert. *The State of the Cross-Domain Nation*. Web 2.0 Security and Privacy (W2SP), 2011.

Besides the SOP, Alloy has been used to model and reason about a variety of systems across different domains—ranging from network protocols, semantic web, bytecode security to electronic voting and medical systems. For many of these systems, Alloy's analysis led to discovery of design flaws and bugs that had eluded the developers, in some cases, for years. We invite our readers to visit the Alloy page[4] and try building a model of their favorite system!

17.10 Appendix: Reusing Modules in Alloy

As mentioned earlier in this chapter, Alloy makes no assumptions about the behavior of the system being modeled. The lack of a built-in paradigm allows the user to encode a wide range of modeling idioms using a small core of the basic language constructs. We could, for example, specify a system as a state machine, a data model with complex invariants, a distributed event model with a global clock, or whatever idiom is most suitable for the problem at hand. Commonly used idioms can be captured as a generic module and reused across multiple systems.

In our model of the SOP, we model the system as a set of endpoints that communicate with each other by making one or more *calls*. Since *call* is a fairly generic notion, we encapsulate its description in a separate Alloy module, to be imported from other modules that rely on it – similar to standard libraries in programming languages:

```
module call[T]
```

In this module declaration, `T` represents a type parameter that can be instantiated to a concrete type that is provided when the module is imported. We will soon see how this type parameter is used.

It is often convenient to describe the system execution as taking place over a global time frame, so that we can talk about calls as occurring before or after each other (or at the same time). To represent the notion of time, we introduce a new signature called `Time`:

```
open util/ordering[Time] as ord
sig Time {}
```

In Alloy, `util/ordering` is a built-in module that imposes a total order on the type parameter, and so by importing `ordering[Time]`, we obtain a set of `Time` objects that behave like other totally ordered sets (e.g., natural numbers).

Note that there is absolutely nothing special about `Time`; we could have named it any other way (for example, `Step` or `State`), and it wouldn't have changed the behavior of the model at all. All we are doing here is using an additional column in a relation as a way of representing the content of a field at different points in a system execution; for example, `cookies` in the `Browser` signature. In this sense, `Time` objects are nothing but helper objects used as a kind of index.

Each call occurs between two points in time—its `start` and `end` times, and is associated with a sender (represented by `from`) and a receiver (`to`):

```
abstract sig Call { start, end: Time, from, to: T }
```

Recall that in our discussion of HTTP requests, we imported the module `call` by passing `Endpoint` as its type parameter. As a result, the parametric type `T` is instantiated to `Endpoint`, and we obtain a set of `Call` objects that are associated with a pair of sender and receiver endpoints. A module can be imported multiple times; for example, we could declare a signature called `UnixProcess`, and instantiate the module `call` to obtain a distinct set of `Call` objects that are sent from one Unix process to another.

[4] `http://alloy.mit.edu`

Web Spreadsheet

Audrey Tang

This chapter introduces a web spreadsheet written in 99 lines of the three languages natively supported by web browsers: HTML, JavaScript, and CSS.

The ES5 version of this project is available as a jsFiddle[1].

19.1 Introduction

When Tim Berners-Lee invented the web in 1990, *web pages* were written in HTML by marking up text with angle-bracketed *tags*, assigning a logical structure to the content. Text marked up within `<a>. . . </a>` became *hyperlinks* that would refer the user to other pages on the web.

In the 1990s, browsers added various presentational tags to the HTML vocabulary, including some notoriously nonstandard tags such as `<blink>. . . </blink>` from Netscape Navigator and `<marquee>. . . </marquee>` from Internet Explorer, causing widespread problems in usability and browser compatibility.

In order to restrict HTML to its original purpose—describing a document's logical structure—browser makers eventually agreed to support two additional languages: CSS to describe presentational styles of a page, and JavaScript (JS) to describe its dynamic interactions.

Since then, the three languages have become more concise and powerful through twenty years of co-evolution. In particular, improvements in JS engines made it practical to deploy large-scale JS frameworks, such as AngularJS[2].

Today, cross-platform *web applications* (such as web spreadsheets) are as ubiquitous and popular as platform-specific applications (such as VisiCalc, Lotus 1-2-3 and Excel) from the previous century.

How many features can a web application offer in 99 lines with AngularJS? Let's see it in action!

19.2 Overview

The spreadsheet[3] directory contains our showcase for late-2014 editions of the three web languages: HTML5[4] for structure, CSS3[5] for presentation, and the JS ES6 "Harmony"[6] standard for interaction.

[1] `http://jsfiddle.net/audreyt/LtDyP/`
[2] `http://angularjs.org/`
[3] `https://github.com/audreyt/500lines/tree/master/spreadsheet/code`
[4] `http://www.w3.org/TR/html5/`
[5] `http://www.w3.org/TR/css3-ui/`
[6] `http://git.io/es6features`

It also uses web storage[7] for data persistence and web workers[8] for running JS code in the background. As of this writing, these web standards are supported by Firefox, Chrome, and Internet Explorer 11+, as well as mobile browsers on iOS 5+ and Android 4+.

Now let's open our spreadsheet[9] in a browser (Figure 19.1):

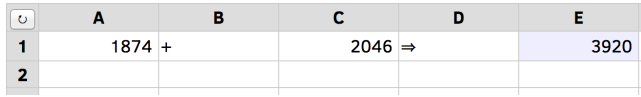

Figure 19.1: Initial Screen

Basic Concepts

The spreadsheet spans two dimensions, with *columns* starting from **A**, and *rows* starting from **1**. Each *cell* has a unique *coordinate* (such as **A1**) and *content* (such as "1874"), which belongs to one of four *types*:

- Text: "+" in **B1** and "->" in **D1**, aligned to the left.
- Number: "1874" in **A1** and "2046" in **C1**, aligned to the right.
- Formula: `=A1+C1` in **E1**, which *calculates* to the *value* "3920", displayed with a light blue background.
- Empty: All cells in row **2** are currently empty.

Click "3920" to set *focus* on **E1**, revealing its formula in an *input box* (Figure 19.2).

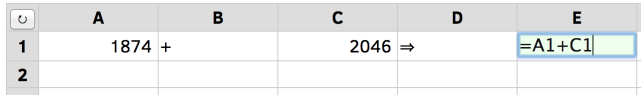

Figure 19.2: Input Box

Now let's set focus on **A1** and *change* its content to "1", causing **E1** to *recalculate* its value to "2047" (Figure 19.3).

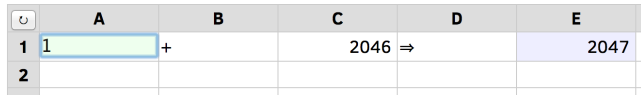

Figure 19.3: Changed Content

Press **ENTER** to set focus to **A2** and change its content to `=Date()`, then press **TAB**, change the content of **B2** to `=alert()`, then press **TAB** again to set focus to `C2` (Figure 19.4).

This shows that a formula may calculate to a number ("2047" in **E1**), a text (the current time in **A2**, aligned to the left), or an *error* (red letters in **B2**, aligned to the center).

Next, let's try entering `=for(;;){}`, the JS code for an infinite loop that never terminates. The spreadsheet will prevent this by automatically *restoring* the content of **C2** after an attempted change.

Now reload the page in the browser with **Ctrl-R** or **Cmd-R** to verify that the spreadsheet content is *persistent*, staying the same across browser sessions. To *reset* the spreadsheet to its original contents, press the 'curved arrow' button on the top-left corner.

[7] `http://www.whatwg.org/specs/web-apps/current-work/multipage/webstorage.html`
[8] `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html`
[9] `http://audreyt.github.io/500lines/spreadsheet/`

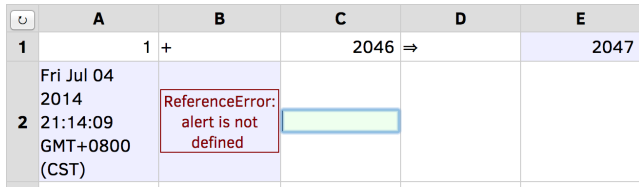

Figure 19.4: Formula Error

Progressive Enhancement

Before we dive into the 99 lines of code, it's worthwhile to disable JS in the browser, reload the page, and note the differences (Figure 19.5).

- Instead of a large grid, only a 2x2 table remains onscreen, with a single content cell.
- Row and column labels are replaced by `{{ row }}` and `{{ col }}`.
- Pressing the reset button produces no effect.
- Pressing **TAB** or clicking into the first line of content still reveals an editable input box.

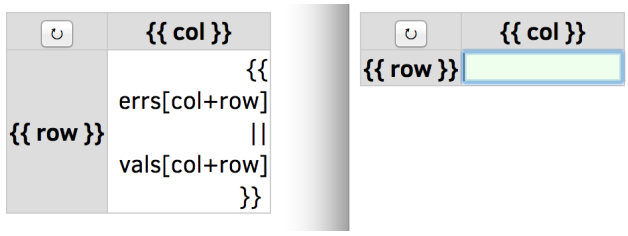

Figure 19.5: With JavaScript Disabled

When we disable the dynamic interactions (JS), the content structure (HTML) and the presentational styles (CSS) remain in effect. If a website is useful with both JS and CSS disabled, we say it adheres to the *progressive enhancement* principle, making its content accessible to the largest audience possible.

Because our spreadsheet is a web application with no server-side code, we must rely on JS to provide the required logic. However, it does work correctly when CSS is not fully supported, such as with screen readers and text-mode browsers.

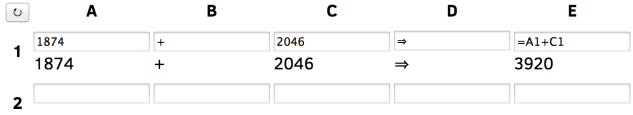

Figure 19.6: With CSS Disabled

As shown in Figure 19.6, if we enable JS in the browser and disable CSS instead, the effects are:

- All background and foreground colors are gone.
- The input box and the cell value are both displayed, instead of just one at a time.
- Otherwise, the application still works the same as the full version.

19.3 Code Walkthrough

Figure 19.7 shows the links between HTML and JS components. In order to make sense of the diagram, let's go through the four source code files, in the same sequence as the browser loads them.

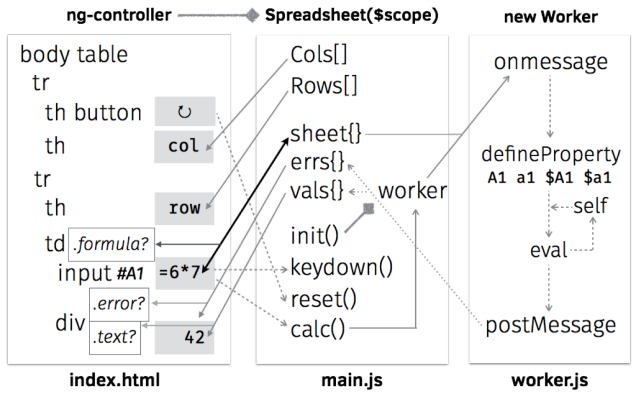

Figure 19.7: Architecture Diagram

- **index.html**: 19 lines
- **main.js**: 38 lines (excluding comments and blank lines)
- **worker.js**: 30 lines (excluding comments and blank lines)
- **styles.css**: 12 lines

HTML

The first line in `index.html` declares that it's written in HTML5 with the UTF-8 encoding:

```
<!DOCTYPE html><html><head><meta charset="UTF-8">
```

Without the `charset` declaration, the browser may display the reset button's Unicode symbol as â†», an example of *mojibake*: garbled text caused by decoding issues.

The next three lines are JS declarations, placed within the `head` section as usual:

```
<script src="lib/angular.js"></script>
<script src="main.js"></script>
<script>
    try { angular.module('500lines') }
    catch(e){ location="es5/index.html" }
</script>
```

The `<script src=". . . ">` tags load JS resources from the same path as the HTML page. For example, if the current URL is `http://abc.com/x/index.html`, then `lib/angular.js` refers to `http://abc.com/x/lib/angular.js`.

The `try{ angular.module('500lines') }` line tests if `main.js` is loaded correctly; if not, it tells the browser to navigate to `es5/index.html` instead. This *redirect-based graceful degradation* technique ensures that for pre-2015 browsers with no ES6 support, we can use the translated-to-ES5 versions of JS programs as a fallback.

The next two lines load the CSS resource, close the head section, and begin the body section containing the user-visible part:

```
  <link href="styles.css" rel="stylesheet">
</head><body ng-app="500lines" ng-controller="Spreadsheet" ng-cloak>
```

The ng-app and ng-controller attributes above tell AngularJS[10] to call the 500lines module's Spreadsheet function, which would return a *model*: an object that provides *bindings* on the document *view*. (The ng-cloak attribute hides the document from display until the bindings are in place.)

As a concrete example, when the user clicks the <button> defined in the next line, its ng-click attribute will trigger and call reset() and calc(), two named functions provided by the JS model:

```
  <table><tr>
    <th><button type="button" ng-click="reset(); calc()"></button></th>
```

The next line uses ng-repeat to display the list of column labels on the top row:

```
    <th ng-repeat="col in Cols">{{ col }}</th>
```

For example, if the JS model defines Cols as ["A","B","C"], then there will be three heading cells (th) labeled accordingly. The {{ col }} notation tells AngularJS to *interpolate* the expression, filling the contents in each th with the current value of col.

Similarly, the next two lines go through values in Rows — [1,2,3] and so on — creating a row for each one and labeling the leftmost th cell with its number:

```
  </tr><tr ng-repeat="row in Rows">
    <th>{{ row }}</th>
```

Because the <tr ng-repeat> tag is not yet closed by </tr> , the row variable is still available for expressions. The next line creates a data cell (td) in the current row and uses both col and row variables in its ng-class attribute:

```
    <td ng-repeat="col in Cols" ng-class="{ formula: ('=' === sheet[col+row][0]) }">
```

A few things are going on here. In HTML, the class attribute describes a *set of class names* that allow CSS to style them differently. The ng-class here evaluates the expression ('=' === sheet[col+row][0]); if it is true, then the <td> gets formula as an additional class, which gives the cell a light-blue background as defined in line 8 of **styles.css** with the .formula *class selector*.

The expression above checks if the current cell is a formula by testing if = is the initial character ([0]) of the string in sheet[col+row], where sheet is a JS model object with coordinates (such as "E1") as properties, and cell contents (such as "=A1+C1") as values. Note that because col is a string and not a number, the + in col+row means concatenation instead of addition.

Inside the <td>, we give the user an input box to edit the cell content stored in sheet[col+row]:

```
      <input id="{{ col+row }}" ng-model="sheet[col+row]" ng-change="calc()"
       ng-model-options="{ debounce: 200 }" ng-keydown="keydown( $event, col, row )">
```

[10] http://angularjs.org/

Here, the key attribute is `ng-model`, which enables a *two-way binding* between the JS model and the input box's editable content. In practice, this means that whenever the user makes a change in the input box, the JS model will update `sheet[col+row]` to match the content, and trigger its `calc()` function to recalculate values of all formula cells.

To avoid repeated calls to `calc()` when the user presses and holds a key, `ng-model-options` limits the update rate to once every 200 milliseconds.

The `id` attribute here is interpolated with the coordinate `col+row`. The `id` attribute of a HTML element must be different from the `id` of all other elements in the same document. This ensures that the `#A1` *ID selector* refers to a single element, instead of a set of elements like the class selector `.formula`. When the user presses the **UP/DOWN/ENTER** keys, the keyboard-navigation logic in `keydown()` will use ID selectors to determine which input box to focus on.

After the input box, we place a `<div>` to display the calculated value of the current cell, represented in the JS model by objects `errs` and `vals`:

```
      <div ng-class="{ error: errs[col+row], text: vals[col+row][0] }">
        {{ errs[col+row] || vals[col+row] }}</div>
```

If an error occurs when computing a formula, the text interpolation uses the error message contained in `errs[col+row]`, and `ng-class` applies the `error` class to the element, allowing CSS to style it differently (with red letters, aligned to the center, etc.).

When there is no error, the `vals[col+row]` on the right side of `||` is interpolated instead. If it's a non-empty string, the initial character (`[0]`) will evaluate to true, applying the `text` class to the element that left-aligns the text.

Because empty strings and numeric values have no initial character, `ng-class` will not assign them any classes, so CSS can style them with right alignment as the default case.

Finally, we close the `ng-repeat` loop in the column level with `</td>`, close the row-level loop with `</tr>`, and end the HTML document with:

```
    </td>
  </tr></table>
</body></html>
```

JS: Main Controller

The `main.js` file defines the `500lines` module and its `Spreadsheet` controller function, as required by the `<body>` element in `index.html`.

As the bridge between the HTML view and the background worker, it has four tasks:

- Define the dimensions and labels of columns and rows.
- Provide event handlers for keyboard navigation and the reset button.
- When the user changes the spreadsheet, send its new content to the worker.
- When computed results arrive from the worker, update the view and save the current state.

The flowchart in Figure 19.8 shows the controller-worker interaction in more detail:

Now let's walk through the code. In the first line, we request the AngularJS `$scope`:

```
angular.module('500lines', []).controller('Spreadsheet', function ($scope, $timeout) {
```

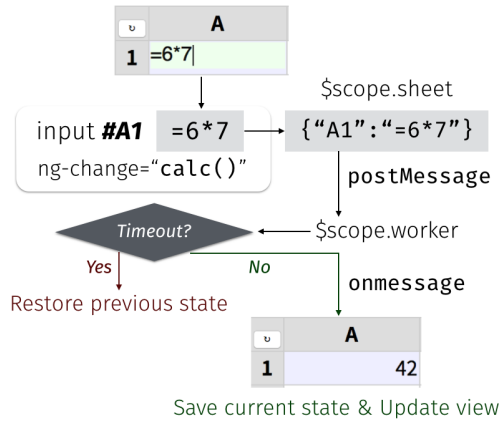

Figure 19.8: Controller-Worker Flowchart

The `$` in `$scope` is part of the variable name. Here we also request the `$timeout`[11] service function from AngularJS; later on, we will use it to prevent infinite-looping formulas.

To put `Cols` and `Rows` into the model, simply define them as properties of `$scope`:

```
// Begin of $scope properties; start with the column/row labels
$scope.Cols = [], $scope.Rows = [];
for (col of range( 'A', 'H' )) { $scope.Cols.push(col); }
for (row of range( 1, 20 )) { $scope.Rows.push(row); }
```

The ES6 for…of[12] syntax makes it easy to loop through ranges with a start and an end point, with the helper function `range` defined as a generator[13]:

```
function* range(cur, end) { while (cur <= end) { yield cur;
```

The `function*` above means that `range` returns an iterator[14], with a `while` loop that would `yield`[15] a single value at a time. Whenever the `for` loop demands the next value, it will resume execution right after the `yield` line:

```
    // If it's a number, increase it by one; otherwise move to next letter
    cur = (isNaN( cur ) ? String.fromCodePoint( cur.codePointAt()+1 ) : cur+1);
} }
```

To generate the next value, we use `isNaN` to see if `cur` is meant as a letter (`NaN` stands for "not a number.") If so, we get the letter's code point value[16], increment it by one, and convert the codepoint[17] back to get its next letter. Otherwise, we simply increase the number by one.

[11] https://docs.angularjs.org/api/ng/service/$timeout
[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of
[13] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*
[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/The_Iterator_protocol
[15] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield
[16] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/codePointAt
[17] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/fromCodePoint

Next up, we define the `keydown()` function that handles keyboard navigation across rows:

```
// UP(38) and DOWN(40)/ENTER(13) move focus to the row above (-1) and below (+1).
$scope.keydown = ({which}, col, row)=>{ switch (which) {
```

The arrow function[18] receives the arguments `($event, col, row)` from `<input ng-keydown>`, using destructuring assignment[19] to assign `$event.which` into the `which` parameter, and checks if it's among the three navigational key codes:

```
    case 38: case 40: case 13: $timeout( ()=>{
```

If it is, we use `$timeout` to schedule a focus change after the current `ng-keydown` and `ng-change` handler. Because `$timeout` requires a function as argument, the `()=>{. . . }` syntax constructs a function to represent the focus-change logic, which starts by checking the direction of movement:

```
      const direction = (which === 38) ? -1 : +1;
```

The `const` declarator means `direction` will not change during the function's execution. The direction to move is either upward (`-1`, from **A2** to **A1**) if the key code is 38 (**UP**), or downward (`+1`, from **A2** to **A3**) otherwise.

Next up, we retrieve the target element using the ID selector syntax (e.g. `"#A3"`), constructed with a template string[20] written in a pair of backticks, concatenating the leading `#`, the current `col` and the target `row + direction`:

```
      const cell = document.querySelector( `#${ col }${ row + direction }` );
      if (cell) { cell.focus(); }
    } );
  } };
```

We put an extra check on the result of `querySelector` because moving upward from **A1** will produce the selector `#A0`, which has no corresponding element, and so will not trigger a focus change — the same goes for pressing **DOWN** at the bottom row.

Next, we define the `reset()` function so the reset button can restore the contents of the `sheet`:

```
// Default sheet content, with some data cells and one formula cell.
$scope.reset = ()=>{
  $scope.sheet = { A1: 1874, B1: '+', C1: 2046, D1: '->', E1: '=A1+C1' }; }
```

The `init()` function tries restoring the `sheet` content from its previous state from the localStorage[21], and defaults to the initial content if it's our first time running the application:

```
// Define the initializer, and immediately call it
($scope.init = ()=>{
  // Restore the previous .sheet; reset to default if it's the first run
  $scope.sheet = angular.fromJson( localStorage.getItem( '' ) );
  if (!$scope.sheet) { $scope.reset(); }
  $scope.worker = new Worker( 'worker.js' );
}).call();
```

[18] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/arrow_functions
[19] https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/1.7\#Pulling_fields_from_objects_passed_as_function_parameter
[20] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/template_strings
[21] https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage\#localStorage

A few things are worth nothing in the `init()` function above:

- We use the `($scope.init = ()=>{. . . }).call()` syntax to define the function and immediately call it.
- Because localStorage only stores strings, we *parse* the `sheet` structure from its JSON[22] representation using `angular.fromJson()`.
- At the last step of `init()`, we create a new web worker[23] thread and assign it to the `worker` scope property. Although the worker is not directly used in the view, it's customary to use `$scope` to share objects used across model functions, in this case between `init()` here and `calc()` below.

While `sheet` holds the user-editable cell content, `errs` and `vals` contain the results of calculations — errors and values — that are read-only to the user:

```
// Formula cells may produce errors in .errs; normal cell contents are in .vals
[$scope.errs, $scope.vals] = [ {}, {} ];
```

With these properties in place, we can define the `calc()` function that triggers whenever the user makes a change to `sheet`:

```
// Define the calculation handler; not calling it yet
$scope.calc = ()=>{
  const json = angular.toJson( $scope.sheet );
```

Here we take a snapshot of the state of `sheet` and store it in the constant `json`, a JSON string. Next up, we construct a `promise` from $timeout[24] that cancels the upcoming computation if it takes more than 99 milliseconds:

```
  const promise = $timeout( ()=>{
    // If the worker has not returned in 99 milliseconds, terminate it
    $scope.worker.terminate();
    // Back up to the previous state and make a new worker
    $scope.init();
    // Redo the calculation using the last-known state
    $scope.calc();
  }, 99 );
```

Since we made sure that `calc()` is called at most once every 200 milliseconds via the `<input ng-model-options>` attribute in HTML, this arrangement leaves 101 milliseconds for `init()` to restore `sheet` to the last known-good state and make a new worker.

The worker's task is to calculate `errs` and `vals` from the contents of `sheet`. Because **main.js** and **worker.js** communicate by message-passing, we need an `onmessage` handler to receive the results once they are ready:

```
  // When the worker returns, apply its effect on the scope
  $scope.worker.onmessage = ({data})=>{
    $timeout.cancel( promise );
    localStorage.setItem( '', json );
    $timeout( ()=>{ [$scope.errs, $scope.vals] = data; } );
  };
```

[22] https://developer.mozilla.org/en-US/docs/Glossary/JSON
[23] https://developer.mozilla.org/en-US/docs/Web/API/Worker
[24] https://docs.angularjs.org/api/ng/service/$timeout

If `onmessage` is called, we know that the `sheet` snapshot in `json` is stable (i.e., containing no infinite-looping formulas), so we cancel the 99-millisecond timeout, write the snapshot to localStorage, and schedule a UI update with a `$timeout` function that updates `errs` and `vals` to the user-visible view.

With the handler in place, we can post the state of `sheet` to the worker, starting its calculation in the background:

```
    // Post the current sheet content for the worker to process
    $scope.worker.postMessage( $scope.sheet );
  };

  // Start calculation when worker is ready
  $scope.worker.onmessage = $scope.calc;
  $scope.worker.postMessage( null );
});
```

JS: Background Worker

There are three reasons for using a web worker to calculate formulas, instead of using the main JS thread for the task:

- While the worker runs in the background, the user is free to continue interacting with the spreadsheet without getting blocked by computation in the main thread.
- Because we accept any JS expression in a formula, the worker provides a *sandbox* that prevents formulas from interfering with the page that contains them, such as by popping out an `alert()` dialog box.
- A formula can refer to any coordinates as variables. The other coordinates may contain another formula that might end in a cyclic reference. To solve this problem, we use the worker's *global scope* object `self`, and define these variables as *getter functions* on `self` to implement the cycle-prevention logic.

With these in mind, let's take a look at the worker's code.

The worker's sole purpose is defining its `onmessage` handler. The handler takes `sheet`, calculates `errs` and `vals`, and posts them back to the main JS thread. We begin by re-initializing the three variables when we receive a message:

```
let sheet, errs, vals;
self.onmessage = ({data})=>{
  [sheet, errs, vals] = [ data, {}, {} ];
```

In order to turn coordinates into global variables, we first iterate over each property in `sheet`, using a `for. . . in` loop:

```
  for (const coord in sheet) {
```

ES6 introduces `const` and `let` declares *block scoped* constants and variables; `const coord` above means that functions defined in the loop would capture the value of `coord` in each iteration.

In contrast, `var coord` in earlier versions of JS would declare a *function scoped* variable, and functions defined in each loop iteration would end up pointing to the same `coord` variable.

Customarily, formula variables are case-insensitive and can optionally have a `$` prefix. Because JS variables are case-sensitive, we use `map` to go over the four variable names for the same coordinate:

```
// Four variable names pointing to the same coordinate: A1, a1, $A1, $a1
[ '', '$' ].map( p => [ coord, coord.toLowerCase() ].map(c => {
  const name = p+c;
```

Note the shorthand arrow function syntax above: `p => ...` is the same as `(p) => { ... }`.

For each variable name, like `A1` and `$a1`, we define an accessor property[25] on `self` that calculates `vals["A1"]` whenever they are evaluated in an expression:

```
// Worker is reused across calculations, so only define each variable once
if ((Object.getOwnPropertyDescriptor( self, name ) || {}).get) { return; }

// Define self['A1'], which is the same thing as the global variable A1
Object.defineProperty( self, name, { get() {
```

The `{ get() { . . . } }` syntax above is shorthand for `{ get: ()=>{ . . . } }`. Because we define only `get` and not `set`, the variables become *read-only* and cannot be modified from user-supplied formulas.

The `get` accessor starts by checking `vals[coord]`, and simply returns it if it's already calculated:

```
if (coord in vals) { return vals[coord]; }
```

If not, we need to calculate `vals[coord]` from `sheet[coord]`.

First we set it to `NaN`, so self-references like setting **A1** to `=A1` will end up with `NaN` instead of an infinite loop:

```
vals[coord] = NaN;
```

Next we check if `sheet[coord]` is a number by converting it to numeric with prefix `+`, assigning the number to `x`, and comparing its string representation with the original string. If they differ, then we set `x` to the original string:

```
// Turn numeric strings into numbers, so =A1+C1 works when both are numbers
let x = +sheet[coord];
if (sheet[coord] !== x.toString()) { x = sheet[coord]; }
```

If the initial character of `x` is `=`, then it's a formula cell. We evaluate the part after `=` with `eval.call()`, using the first argument `null` to tell `eval` to run in the *global scope*, hiding the *lexical scope* variables like `x` and `sheet` from the evaluation:

```
// Evaluate formula cells that begin with =
try { vals[coord] = (('=' === x[0]) ? eval.call( null, x.slice( 1 ) ) : x);
```

If the evaluation succeeds, the result is stored into `vals[coord]`. For non-formula cells, the value of `vals[coord]` is simply `x`, which may be a number or a string.

If `eval` results in an error, the `catch` block tests if it's because the formula refers to an empty cell not yet defined in `self`:

```
} catch (e) {
  const match = /\$?[A-Za-z]+[1-9][0-9]*\b/.exec( e );
  if (match && !( match[0] in self )) {
```

[25] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

In that case, we set the missing cell's default value to "0", clear `vals[coord]`, and re-run the current computation using `self[coord]`:

```
            // The formula refers to a uninitialized cell; set it to 0 and retry
            self[match[0]] = 0;
            delete vals[coord];
            return self[coord];
          }
```

If the user gives the missing cell a content later on in `sheet[coord]`, then the temporary value would be overridden by `Object.defineProperty`.

Other kinds of errors are stored in `errs[coord]`:

```
          // Otherwise, stringify the caught exception in the errs object
          errs[coord] = e.toString();
        }
```

In case of errors, the value of `vals[coord]` will remain `NaN` because the assignment did not finish executing.

Finally, the `get` accessor returns the calculated value stored in `vals[coord]`, which must be a number, a Boolean value, or a string:

```
        // Turn vals[coord] into a string if it's not a number or Boolean
        switch (typeof vals[coord]) {
            case 'function': case 'object': vals[coord]+='';
        }
        return vals[coord];
      } } );
    }));
  }
```

With accessors defined for all coordinates, the worker goes through the coordinates again, invoking each accessor with `self[coord]`, then posts the resulting `errs` and `vals` back to the main JS thread:

```
  // For each coordinate in the sheet, call the property getter defined above
  for (const coord in sheet) { self[coord]; }
  return [ errs, vals ];
}
```

CSS

The **styles.css** file contains just a few selectors and their presentational styles. First, we style the table to merge all cell borders together, leaving no spaces between neighboring cells:

```
table { border-collapse: collapse; }
```

Both the heading and data cells share the same border style, but we can tell them apart by their background colors: heading cells are light gray, data cells are white by default, and formula cells get a light blue background:

```
th, td { border: 1px solid #ccc; }
th { background: #ddd; }
td.formula { background: #eef; }
```

The displayed width is fixed for each cell's calculated values. Empty cells receive a minimal height, and long lines are clipped with a trailing ellipsis:

```
td div { text-align: right; width: 120px; min-height: 1.2em;
         overflow: hidden; text-overflow: ellipsis; }
```

The text alignment and decorations are determined by each value's type, as reflected by the `text` and `error` class selectors:

```
div.text { text-align: left; }
div.error { text-align: center; color: #800; font-size: 90%; border: solid 1px #800 }
```

As for the user-editable `input` box, we use *absolute positioning* to overlay it on top of its cell, and make it transparent so the underlying `div` with the cell's value shows through:

```
input { position: absolute; border: 0; padding: 0;
        width: 120px; height: 1.3em; font-size: 100%;
        color: transparent; background: transparent; }
```

When the user sets focus on the input box, it springs into the foreground:

```
input:focus { color: #111; background: #efe; }
```

Furthermore, the underlying `div` is collapsed into a single line, so it's completely covered by the input box:

```
input:focus + div { white-space: nowrap; }
```

19.4 Conclusion

Since this book is *500 Lines or Less*, a web spreadsheet in 99 lines is a minimal example—please feel free to experiment and extend it in any direction you'd like.

Here are some ideas, all easily reachable in the remaining space of 401 lines:

- A collaborative online editor using ShareJS[26], AngularFire[27] or GoAngular[28].
- Markdown syntax support for text cells, using angular-marked[29].
- Common formula functions (`SUM`, `TRIM`, etc.) from the OpenFormula standard[30].
- Interoperate with popular spreadsheet formats, such as CSV and SpreadsheetML via SheetJS[31].
- Import from and export to online spreadsheet services, such as Google Spreadsheet and EtherCalc[32].

[26] `http://sharejs.org/`
[27] `http://angularfire.com`
[28] `http://goangular.org/`
[29] `http://ngmodules.org/modules/angular-marked`
[30] `https://en.wikipedia.org/wiki/OpenFormula`
[31] `http://sheetjs.com/`
[32] `http://ethercalc.net/`

A Note on JS versions

This chapter aims to demonstrate new concepts in ES6, so we use the Traceur compiler[33] to translate source code to ES5 to run on pre-2015 browsers.

If you prefer to work directly with the 2010 edition of JS, the as-javascript-1.8.5[34] directory has **main.js** and **worker.js** written in the style of ES5; the source code[35] is line-by-line comparable to the ES6 version with the same line count.

For people preferring a cleaner syntax, the as-livescript-1.3.0[36] directory uses LiveScript[37] instead of ES6 to write **main.ls** and **worker.ls**; it is 20 lines shorter[38] than the JS version.

Building on the LiveScript language, the as-react-livescript[39] directory uses the ReactJS[40] framework; it is 10 lines more longer[41] than the AngularJS equivalent, but runs considerably faster.

If you are interested in translating this example to alternate JS languages, send a pull request[42]—I'd love to hear about it!

[33] https://github.com/google/traceur-compiler
[34] https://audreyt.github.io/500lines/spreadsheet/as-javascript-1.8.5/
[35] https://github.com/audreyt/500lines/tree/master/spreadsheet/as-javascript-1.8.5
[36] https://audreyt.github.io/500lines/spreadsheet/as-livescript-1.3.0/
[37] http://livescript.net/
[38] https://github.com/audreyt/500lines/tree/master/spreadsheet/as-livescript-1.3.0
[39] https://audreyt.github.io/500lines/spreadsheet/as-react-livescript/
[40] https://facebook.github.io/react/
[41] https://github.com/audreyt/500lines/tree/master/spreadsheet/as-react-livescript
[42] https://github.com/audreyt/500lines/pulls

A Template Engine

Ned Batchelder

21.1 Introduction

Most programs contain a lot of logic, and a little bit of literal textual data. Programming languages are designed to be good for this sort of programming. But some programming tasks involve only a little bit of logic, and a great deal of textual data. For these tasks, we'd like to have a tool better suited to these text-heavy problems. A template engine is such a tool. In this chapter, we build a simple template engine.

The most common example of one of these text-heavy tasks is in web applications. An important phase in any web application is generating HTML to be served to the browser. Very few HTML pages are completely static: they involve at least a small amount of dynamic data, such as the user's name. Usually, they contain a great deal of dynamic data: product listings, friends' news updates, and so on.

At the same time, every HTML page contains large swaths of static text. And these pages are large, containing tens of thousands of bytes of text. The web application developer has a problem to solve: how best to generate a large string containing a mix of static and dynamic data? To add to the problem, the static text is actually HTML markup that is authored by another member of the team, the front-end designer, who wants to be able to work with it in familiar ways.

For purposes of illustration, let's imagine we want to produce this toy HTML:

```
<p>Welcome, Charlie!</p>
<p>Products:</p>
<ul>
    <li>Apple: $1.00</li>
    <li>Fig: $1.50</li>
    <li>Pomegranate: $3.25</li>
</ul>
```

Here, the user's name will be dynamic, as will the names and prices of the products. Even the number of products isn't fixed: at another moment, there could be more or fewer products to display.

One way to make this HTML would be to have string constants in our code, and join them together to produce the page. Dynamic data would be inserted with string substitution of some sort. Some of our dynamic data is repetitive, like our lists of products. This means we'll have chunks of HTML that repeat, so those will have to be handled separately and combined with the rest of the page.

Producing our toy page in this way might look like this:

```
# The main HTML for the whole page.
PAGE_HTML = """
<p>Welcome, {name}!</p>
<p>Products:</p>
<ul>
{products}
</ul>
"""

# The HTML for each product displayed.
PRODUCT_HTML = "<li>{prodname}: {price}</li>\n"

def make_page(username, products):
    product_html = ""
    for prodname, price in products:
        product_html += PRODUCT_HTML.format(
            prodname=prodname, price=format_price(price))
    html = PAGE_HTML.format(name=username, products=product_html)
    return html
```

This works, but we have a mess on our hands. The HTML is in multiple string constants embedded in our application code. The logic of the page is hard to see because the static text is broken into separate pieces. The details of how data is formatted is lost in the Python code. In order to modify the HTML page, our front-end designer would need to be able to edit Python code to make HTML changes. Imagine what the code would look like if the page were ten (or one hundred) times more complicated; it would quickly become unworkable.

21.2 Templates

The better way to produce HTML pages is with *templates*. The HTML page is authored as a template, meaning that the file is mostly static HTML, with dynamic pieces embedded in it using special notation. Our toy page above could look like this as a template:

```
<p>Welcome, {{user_name}}!</p>
<p>Products:</p>
<ul>
{% for product in product_list %}
    <li>{{ product.name }}:
        {{ product.price|format_price }}</li>
{% endfor %}
</ul>
```

Here the focus is on the HTML text, with logic embedded in the HTML. Contrast this document-centric approach with our logic-centric code above. Our earlier program was mostly Python code, with HTML embedded in the Python logic. Here our program is mostly static HTML markup.

The mostly-static style used in templates is the opposite of how most programming languages work. For example, with Python, most of the source file is executable code, and if you need literal static text, you embed it in a string literal:

```
def hello():
    print("Hello, world!")

hello()
```

When Python reads this source file, it interprets text like `def hello():` as instructions to be executed. The double quote character in `print("Hello, world!")` indicates that the following text is meant literally, until the closing double quote. This is how most programming languages work: mostly dynamic, with some static pieces embedded in the instructions. The static pieces are indicated by the double-quote notation.

A template language flips this around: the template file is mostly static literal text, with special notation to indicate the executable dynamic parts.

```
<p>Welcome, {{user_name}}!</p>
```

Here the text is meant to appear literally in the resulting HTML page, until the '`{{`' indicates a switch into dynamic mode, where the `user_name` variable will be substituted into the output.

String formatting functions such as Python's `"foo = {foo}!".format(foo=17)` are examples of mini-languages used to create text from a string literal and the data to be inserted. Templates extend this idea to include constructs like conditionals and loops, but the difference is only of degree.

These files are called templates because they are used to produce many pages with similar structure but differing details.

To use HTML templates in our programs, we need a *template engine*: a function that takes a static template describing the structure and static content of the page, and a dynamic *context* that provides the dynamic data to plug into the template. The template engine combines the template and the context to produce a complete string of HTML. The job of a template engine is to interpret the template, replacing the dynamic pieces with real data.

By the way, there's often nothing particular about HTML in a template engine, it could be used to produce any textual result. For example, they are also used to produce plain-text email messages. But usually they are used for HTML, and occasionally have HTML-specific features, such as escaping, which makes it possible to insert values into the HTML without worrying about which characters are special in HTML.

21.3 Supported Syntax

Template engines vary in the syntax they support. Our template syntax is based on Django, a popular web framework. Since we are implementing our engine in Python, some Python concepts will appear in our syntax. We've already seen some of this syntax in our toy example at the top of the chapter, but this is a quick summary of all of the syntax we'll implement.

Data from the context is inserted using double curly braces:

```
<p>Welcome, {{user_name}}!</p>
```

The data available to the template is provided in the context when the template is rendered. More on that later.

Template engines usually provide access to elements within data using a simplified and relaxed syntax. In Python, these expressions all have different effects:

```
dict["key"]
obj.attr
obj.method()
```

In our template syntax, all of these operations are expressed with a dot:

```
dict.key
obj.attr
obj.method
```

The dot will access object attributes or dictionary values, and if the resulting value is callable, it's automatically called. This is different than the Python code, where you need to use different syntax for those operations. This results in simpler template syntax:

```
<p>The price is: {{product.price}}, with a {{product.discount}}% discount.</p>
```

You can use functions called *filters* to modify values. Filters are invoked with a pipe character:

```
<p>Short name: {{story.subject|slugify|lower}}</p>
```

Building interesting pages usually requires at least a small amount of decision-making, so conditionals are available:

```
{% if user.is_logged_in %}
    <p>Welcome, {{ user.name }}!</p>
{% endif %}
```

Looping lets us include collections of data in our pages:

```
<p>Products:</p>
<ul>
{% for product in product_list %}
    <li>{{ product.name }}: {{ product.price|format_price }}</li>
{% endfor %}
</ul>
```

As with other programming languages, conditionals and loops can be nested to build complex logical structures.

Lastly, so that we can document our templates, comments appear between brace-hashes:

```
{# This is the best template ever! #}
```

21.4 Implementation Approaches

In broad strokes, the template engine will have two main phases: *parsing* the template, and then *rendering* the template.

Rendering the template specifically involves:

- Managing the dynamic context, the source of the data
- Executing the logic elements
- Implementing dot access and filter execution

The question of what to pass from the parsing phase to the rendering phase is key. What does parsing produce that can be rendered? There are two main options; we'll call them *interpretation* and *compilation*, using the terms loosely from other language implementations.

In an interpretation model, parsing produces a data structure representing the structure of the template. The rendering phase walks that data structure, assembling the result text based on the instructions it finds. For a real-world example, the Django template engine uses this approach.

In a compilation model, parsing produces some form of directly executable code. The rendering phase executes that code, producing the result. Jinja2 and Mako are two examples of template engines that use the compilation approach.

Our implementation of the engine uses compilation: we compile the template into Python code. When run, the Python code assembles the result.

The template engine described here was originally written as part of coverage.py, to produce HTML reports. In coverage.py, there are only a few templates, and they are used over and over to produce many files from the same template. Overall, the program ran faster if the templates were compiled to Python code, because even though the compilation process was a bit more complicated, it only had to run once, while the execution of the compiled code ran many times, and was faster than interpreting a data structure many times.

It's a bit more complicated to compile the template to Python, but it's not as bad as you might think. And besides, as any developer can tell you, it's more fun to write a program to write a program than it is to write a program!

Our template compiler is a small example of a general technique called code generation. Code generation underlies many powerful and flexible tools, including programming language compilers. Code generation can get complex, but is a useful technique to have in your toolbox.

Another application of templates might prefer the interpreted approach, if templates will be used only a few times each. Then the effort to compile to Python won't pay off in the long run, and a simpler interpretation process might perform better overall.

21.5 Compiling to Python

Before we get to the code of the template engine, let's look at the code it produces. The parsing phase will convert a template into a Python function. Here is our small template again:

```
<p>Welcome, {{user_name}}!</p>
<p>Products:</p>
<ul>
{% for product in product_list %}
    <li>{{ product.name }}:
        {{ product.price|format_price }}</li>
{% endfor %}
</ul>
```

Our engine will compile this template to Python code. The resulting Python code looks unusual, because we've chosen some shortcuts that produce slightly faster code. Here is the Python (slightly reformatted for readability):

```
def render_function(context, do_dots):
    c_user_name = context['user_name']
    c_product_list = context['product_list']
```

```
    c_format_price = context['format_price']

    result = []
    append_result = result.append
    extend_result = result.extend
    to_str = str

    extend_result([
        '<p>Welcome, ',
        to_str(c_user_name),
        '!</p>\n<p>Products:</p>\n<ul>\n'
    ])
    for c_product in c_product_list:
        extend_result([
            '\n    <li>',
            to_str(do_dots(c_product, 'name')),
            ':\n        ',
            to_str(c_format_price(do_dots(c_product, 'price'))),
            '</li>\n'
        ])
    append_result('\n</ul>\n')
    return ''.join(result)
```

Each template is converted into a `render_function` function that takes a dictionary of data called the context. The body of the function starts by unpacking the data from the context into local names, because they are faster for repeated use. All the context data goes into locals with a `c_` prefix so that we can use other local names without fear of collisions.

The result of the template will be a string. The fastest way to build a string from parts is to create a list of strings, and join them together at the end. `result` will be the list of strings. Because we're going to add strings to this list, we capture its `append` and `extend` methods in the local names `result_append` and `result_extend`. The last local we create is a `to_str` shorthand for the `str` built-in.

These kinds of shortcuts are unusual. Let's look at them more closely. In Python, a method call on an object like `result.append("hello")` is executed in two steps. First, the append attribute is fetched from the result object: `result.append`. Then the value fetched is invoked as a function, passing it the argument `"hello"`. Although we're used to seeing those steps performed together, they really are separate. If you save the result of the first step, you can perform the second step on the saved value. So these two Python snippets do the same thing:

```
# The way we're used to seeing it:
result.append("hello")

# But this works the same:
append_result = result.append
append_result("hello")
```

In the template engine code, we've split it out this way so that we only do the first step once, no matter how many times we do the second step. This saves us a small amount of time, because we avoid taking the time to look up the append attribute.

This is an example of a micro-optimization: an unusual coding technique that gains us tiny improvements in speed. Micro-optimizations can be less readable, or more confusing, so they are

only justified for code that is a proven performance bottleneck. Developers disagree on how much micro-optimization is justified, and some beginners overdo it. The optimizations here were added only after timing experiments showed that they improved performance, even if only a little bit. Micro-optimizations can be instructive, as they make use of some exotic aspects of Python, but don't over-use them in your own code.

The shortcut for `str` is also a micro-optimization. Names in Python can be local to a function, global to a module, or built-in to Python. Looking up a local name is faster than looking up a global or a built-in. We're used to the fact that `str` is a builtin that is always available, but Python still has to look up the name `str` each time it is used. Putting it in a local saves us another small slice of time because locals are faster than builtins.

Once those shortcuts are defined, we're ready for the Python lines created from our particular template. Strings will be added to the result list using the `append_result` or `extend_result` shorthands, depending on whether we have one string to add, or more than one. Literal text in the template becomes a simple string literal.

Having both append and extend adds complexity, but remember we're aiming for the fastest execution of the template, and using extend for one item means making a new list of one item so that we can pass it to extend.

Expressions in `{{ ... }}` are computed, converted to strings, and added to the result. Dots in the expression are handled by the `do_dots` function passed into our function, because the meaning of the dotted expressions depends on the data in the context: it could be attribute access or item access, and it could be a callable.

The logical structures `{% if ... %}` and `{% for ... %}` are converted into Python conditionals and loops. The expression in the `{% if/for ... %}` tag will become the expression in the `if` or `for` statement, and the contents up until the `{% end... %}` tag will become the body of the statement.

21.6 Writing the Engine

Now that we understand what the engine will do, let's walk through the implementation.

The Templite class

The heart of the template engine is the Templite class. (Get it? It's a template, but it's lite!)

The Templite class has a small interface. You construct a Templite object with the text of the template, then later you can use the `render` method on it to render a particular context, the dictionary of data, through the template:

```
# Make a Templite object.
templite = Templite('''
    <h1>Hello {{name|upper}}!</h1>
    {% for topic in topics %}
        <p>You are interested in {{topic}}.</p>
    {% endfor %}
    ''',
    {'upper': str.upper},
)

# Later, use it to render some data.
text = templite.render({
```

```
    'name': "Ned",
    'topics': ['Python', 'Geometry', 'Juggling'],
})
```

We pass the text of the template when the object is created so that we can do the compile step just once, and later call `render` many times to reuse the compiled results.

The constructor also accepts a dictionary of values, an initial context. These are stored in the Templite object, and will be available when the template is later rendered. These are good for defining functions or constants we want to be available everywhere, like `upper` in the previous example.

Before we discuss the implementation of Templite, we have a helper to define first: CodeBuilder.

CodeBuilder

The bulk of the work in our engine is parsing the template and producing the necessary Python code. To help with producing the Python, we have the CodeBuilder class, which handles the bookkeeping for us as we construct the Python code. It adds lines of code, manages indentation, and finally gives us values from the compiled Python.

One CodeBuilder object is responsible for a complete chunk of Python code. As used by our template engine, the chunk of Python is always a single complete function definition. But the CodeBuilder class makes no assumption that it will only be one function. This keeps the CodeBuilder code more general, and less coupled to the rest of the template engine code.

As we'll see, we also use nested CodeBuilders to make it possible to put code at the beginning of the function even though we don't know what it will be until we are nearly done.

A CodeBuilder object keeps a list of strings that will together be the final Python code. The only other state it needs is the current indentation level:

```
class CodeBuilder(object):
    """Build source code conveniently."""

    def __init__(self, indent=0):
        self.code = []
        self.indent_level = indent
```

CodeBuilder doesn't do much. `add_line` adds a new line of code, which automatically indents the text to the current indentation level, and supplies a newline:

```
    def add_line(self, line):
        """Add a line of source to the code.

        Indentation and newline will be added for you, don't provide them.

        """
        self.code.extend([" " * self.indent_level, line, "\n"])
```

`indent` and `dedent` increase or decrease the indentation level:

```
    INDENT_STEP = 4      # PEP8 says so!

    def indent(self):
        """Increase the current indent for following lines."""
```

```
        self.indent_level += self.INDENT_STEP

    def dedent(self):
        """Decrease the current indent for following lines."""
        self.indent_level -= self.INDENT_STEP
```

`add_section` is managed by another CodeBuilder object. This lets us keep a reference to a place in the code, and add text to it later. The `self.code` list is mostly a list of strings, but will also hold references to these sections:

```
    def add_section(self):
        """Add a section, a sub-CodeBuilder."""
        section = CodeBuilder(self.indent_level)
        self.code.append(section)
        return section
```

`__str__` produces a single string with all the code. This simply joins together all the strings in `self.code`. Note that because `self.code` can contain sections, this might call other `CodeBuilder` objects recursively:

```
    def __str__(self):
        return "".join(str(c) for c in self.code)
```

`get_globals` yields the final values by executing the code. This stringifies the object, executes it to get its definitions, and returns the resulting values:

```
    def get_globals(self):
        """Execute the code, and return a dict of globals it defines."""
        # A check that the caller really finished all the blocks they started.
        assert self.indent_level == 0
        # Get the Python source as a single string.
        python_source = str(self)
        # Execute the source, defining globals, and return them.
        global_namespace = {}
        exec(python_source, global_namespace)
        return global_namespace
```

This last method uses some exotic features of Python. The `exec` function executes a string containing Python code. The second argument to `exec` is a dictionary that will collect up the globals defined by the code. So for example, if we do this:

```
python_source = """\
SEVENTEEN = 17

def three():
    return 3
"""
global_namespace = {}
exec(python_source, global_namespace)
```

then `global_namespace['SEVENTEEN']` is 17, and `global_namespace['three']` is an actual function named `three`.

Although we only use CodeBuilder to produce one function, there's nothing here that limits it to that use. This makes the class simpler to implement, and easier to understand.

CodeBuilder lets us create a chunk of Python source code, and has no specific knowledge about our template engine at all. We could use it in such a way that three different functions would be defined in the Python, and then `get_globals` would return a dict of three values, the three functions. As it happens, our template engine only needs to define one function. But it's better software design to keep that implementation detail in the template engine code, and out of our CodeBuilder class.

Even as we're actually using it—to define a single function—having `get_globals` return the dictionary keeps the code more modular because it doesn't need to know the name of the function we've defined. Whatever function name we define in our Python source, we can retrieve that name from the dict returned by `get_globals`.

Now we can get into the implementation of the Templite class itself, and see how and where CodeBuilder is used.

The Templite class implementation

Most of our code is in the Templite class. As we've discussed, it has both a compilation and a rendering phase.

Compiling

All of the work to compile the template into a Python function happens in the Templite constructor. First the contexts are saved away:

```
    def __init__(self, text, *contexts):
        """Construct a Templite with the given `text`.

        `contexts` are dictionaries of values to use for future renderings.
        These are good for filters and global values.

        """
        self.context = {}
        for context in contexts:
            self.context.update(context)
```

Notice we used `*contexts` as the parameter. The asterisk denotes that any number of positional arguments will be packed into a tuple and passed in as `contexts`. This is called argument unpacking, and means that the caller can provide a number of different context dictionaries. Now any of these calls are valid:

```
t = Templite(template_text)
t = Templite(template_text, context1)
t = Templite(template_text, context1, context2)
```

The context arguments (if any) are supplied to the constructor as a tuple of contexts. We can then iterate over the `contexts` tuple, dealing with each of them in turn. We simply create one combined dictionary called `self.context` which has the contents of all of the supplied contexts. If duplicate names are provided in the contexts, the last one wins.

To make our compiled function as fast as possible, we extract context variables into Python locals. We'll get those names by keeping a set of variable names we encounter, but we also need to track the names of variables defined in the template, the loop variables:

```
self.all_vars = set()
self.loop_vars = set()
```

Later we'll see how these get used to help construct the prologue of our function. First, we'll use the CodeBuilder class we wrote earlier to start to build our compiled function:

```
code = CodeBuilder()

code.add_line("def render_function(context, do_dots):")
code.indent()
vars_code = code.add_section()
code.add_line("result = []")
code.add_line("append_result = result.append")
code.add_line("extend_result = result.extend")
code.add_line("to_str = str")
```

Here we construct our CodeBuilder object, and start writing lines into it. Our Python function will be called `render_function`, and will take two arguments: `context` is the data dictionary it should use, and `do_dots` is a function implementing dot attribute access.

The context here is the combination of the data context passed to the Templite constructor, and the data context passed to the render function. It's the complete set of data available to the template that we made in the Templite constructor.

Notice that CodeBuilder is very simple: it doesn't "know" about function definitions, just lines of code. This keeps CodeBuilder simple, both in its implementation, and in its use. We can read our generated code here without having to mentally interpolate too many specialized CodeBuilder.

We create a section called `vars_code`. Later we'll write the variable extraction lines into that section. The `vars_code` object lets us save a place in the function that can be filled in later when we have the information we need.

Then four fixed lines are written, defining a result list, shortcuts for the methods to append to or extend that list, and a shortcut for the `str()` builtin. As we discussed earlier, this odd step squeezes just a little bit more performance out of our rendering function.

The reason we have both the `append` and the `extend` shortcut is so we can use the most effective method, depending on whether we have one line to add to our result, or more than one.

Next we define an inner function to help us with buffering output strings:

```
buffered = []
def flush_output():
    """Force `buffered` to the code builder."""
    if len(buffered) == 1:
        code.add_line("append_result(%s)" % buffered[0])
    elif len(buffered) > 1:
        code.add_line("extend_result([%s])" % ", ".join(buffered))
    del buffered[:]
```

As we create chunks of output that need to go into our compiled function, we need to turn them into function calls that append to our result. We'd like to combine repeated append calls into one extend call. This is another micro-optimization. To make this possible, we buffer the chunks.

The `buffered` list holds strings that are yet to be written to our function source code. As our template compilation proceeds, we'll append strings to `buffered`, and flush them to the function source when we reach control flow points, like if statements, or the beginning or ends of loops.

The `flush_output` function is a *closure*, which is a fancy word for a function that refers to variables outside of itself. Here `flush_output` refers to `buffered` and `code`. This simplifies our calls to the function: we don't have to tell `flush_output` what buffer to flush, or where to flush it; it knows all that implicitly.

If only one string has been buffered, then the `append_result` shortcut is used to append it to the result. If more than one is buffered, then the `extend_result` shortcut is used, with all of them, to add them to the result. Then the buffered list is cleared so more strings can be buffered.

The rest of the compiling code will add lines to the function by appending them to `buffered`, and eventually call `flush_output` to write them to the CodeBuilder.

With this function in place, we can have a line of code in our compiler like this:

```
buffered.append("'hello'")
```

which will mean that our compiled Python function will have this line:

```
append_result('hello')
```

which will add the string `hello` to the rendered output of the template. We have multiple levels of abstraction here which can be difficult to keep straight. The compiler uses `buffered.append("'hello'")`, which creates `append_result('hello')` in the compiled Python function, which when run, appends `hello` to the template result.

Back to our Templite class. As we parse control structures, we want to check that they are properly nested. The `ops_stack` list is a stack of strings:

```
ops_stack = []
```

When we encounter an `{% if .. %}` tag (for example), we'll push `'if'` onto the stack. When we find an `{% endif %}` tag, we can pop the stack and report an error if there was no `'if'` at the top of the stack.

Now the real parsing begins. We split the template text into a number of tokens using a regular expression, or *regex*. Regexes can be daunting: they are a very compact notation for complex pattern matching. They are also very efficient, since the complexity of matching the pattern is implemented in C in the regular expression engine, rather than in your own Python code. Here's our regex:

```
tokens = re.split(r"(?s)({{.*?}}|{%.*?%}|{#.*?#})", text)
```

This looks complicated; let's break it down.

The `re.split` function will split a string using a regex. Our pattern is parenthesized, so the matches will be used to split the string, and will also be returned as pieces in the split list. Our pattern will match our tag syntaxes, but we've parenthesized it so that the string will be split at the tags, and the tags will also be returned.

The `(?s)` flag in the regex means that a dot should match even a newline. Next we have our parenthesized group of three alternatives: `{{.*?}}` matches an expression, `{%.*?%}` matches a tag, and `{#.*?#}` matches a comment. In all of these, we use `.*?` to match any number of characters, but the shortest sequence that matches.

The result of `re.split` is a list of strings. For example, this template text:

```
<p>Topics for {{name}}: {% for t in topics %}{{t}}, {% endfor %}</p>
```

would be split into these pieces:

```
[
    '<p>Topics for ',               # literal
    '{{name}}',                     # expression
    ': ',                           # literal
    '{% for t in topics %}',        # tag
    '',                             # literal (empty)
    '{{t}}',                        # expression
    ', ',                           # literal
    '{% endfor %}',                 # tag
    '</p>'                          # literal
]
```

Once the text is split into tokens like this, we can loop over the tokens, and deal with each in turn. By splitting them according to their type, we can handle each type separately.

The compilation code is a loop over these tokens:

```
    for token in tokens:
```

Each token is examined to see which of the four cases it is. Just looking at the first two characters is enough. The first case is a comment, which is easy to handle: just ignore it and move on to the next token:

```
        if token.startswith('{#'):
            # Comment: ignore it and move on.
            continue
```

For the case of `{{...}}` expressions, we cut off the two braces at the front and back, strip off the white space, and pass the entire expression to `_expr_code`:

```
        elif token.startswith('{{'):
            # An expression to evaluate.
            expr = self._expr_code(token[2:-2].strip())
            buffered.append("to_str(%s)" % expr)
```

The `_expr_code` method will compile the template expression into a Python expression. We'll see that function later. We use the `to_str` function to force the expression's value to be a string, and add that to our result.

The third case is the big one: `{% ... %}` tags. These are control structures that will become Python control structures. First we have to flush our buffered output lines, then we extract a list of words from the tag:

```
        elif token.startswith('{%'):
            # Action tag: split into words and parse further.
            flush_output()
            words = token[2:-2].strip().split()
```

Now we have three sub-cases, based on the first word in the tag: `if`, `for`, or `end`. The `if` case shows our simple error handling and code generation:

```
if words[0] == 'if':
    # An if statement: evaluate the expression to determine if.
    if len(words) != 2:
        self._syntax_error("Don't understand if", token)
    ops_stack.append('if')
    code.add_line("if %s:" % self._expr_code(words[1]))
    code.indent()
```

The `if` tag should have a single expression, so the `words` list should have only two elements in it. If it doesn't, we use the `_syntax_error` helper method to raise a syntax error exception. We push `'if'` onto `ops_stack` so that we can check the `endif` tag. The expression part of the `if` tag is compiled to a Python expression with `_expr_code`, and is used as the conditional expression in a Python `if` statement.

The second tag type is `for`, which will be compiled to a Python `for` statement:

```
elif words[0] == 'for':
    # A loop: iterate over expression result.
    if len(words) != 4 or words[2] != 'in':
        self._syntax_error("Don't understand for", token)
    ops_stack.append('for')
    self._variable(words[1], self.loop_vars)
    code.add_line(
        "for c_%s in %s:" % (
            words[1],
            self._expr_code(words[3])
        )
    )
    code.indent()
```

We do a check of the syntax and push `'for'` onto the stack. The `_variable` method checks the syntax of the variable, and adds it to the set we provide. This is how we collect up the names of all the variables during compilation. Later we'll need to write the prologue of our function, where we'll unpack all the variable names we get from the context. To do that correctly, we need to know the names of all the variables we encountered, `self.all_vars`, and the names of all the variables defined by loops, `self.loop_vars`.

We add one line to our function source, a `for` statement. All of our template variables are turned into Python variables by prepending `c_` to them, so that we know they won't collide with other names we're using in our Python function. We use `_expr_code` to compile the iteration expression from the template into an iteration expression in Python.

The last kind of tag we handle is an `end` tag; either `{% endif %}` or `{% endfor %}`. The effect on our compiled function source is the same: simply unindent to end the `if` or `for` statement that was started earlier:

```
elif words[0].startswith('end'):
    # Endsomething.  Pop the ops stack.
    if len(words) != 1:
        self._syntax_error("Don't understand end", token)
    end_what = words[0][3:]
    if not ops_stack:
        self._syntax_error("Too many ends", token)
```

```
                start_what = ops_stack.pop()
                if start_what != end_what:
                    self._syntax_error("Mismatched end tag", end_what)
                code.dedent()
```

Notice here that the actual work needed for the end tag is one line: unindent the function source. The rest of this clause is all error checking to make sure that the template is properly formed. This isn't unusual in program translation code.

Speaking of error handling, if the tag isn't an `if`, a `for`, or an `end`, then we don't know what it is, so raise a syntax error:

```
            else:
                self._syntax_error("Don't understand tag", words[0])
```

We're done with the three different special syntaxes (`{{...}}`, `{#...#}`, and `{%...%}`). What's left is literal content. We'll add the literal string to the buffered output, using the `repr` built-in function to produce a Python string literal for the token:

```
        else:
            # Literal content.  If it isn't empty, output it.
            if token:
                buffered.append(repr(token))
```

If we didn't use `repr`, then we'd end up with lines like this in our compiled function:

```
append_result(abc)      # Error! abc isn't defined
```

We need the value to be quoted like this:

```
append_result('abc')
```

The `repr` function supplies the quotes around the string for us, and also provides backslashes where needed:

```
append_result('"Don\'t you like my hat?" he asked.')
```

Notice that we first check if the token is an empty string with `if token:`, since there's no point adding an empty string to the output. Because our regex is splitting on tag syntax, adjacent tags will have an empty token between them. The check here is an easy way to avoid putting useless `append_result("")` statements into our compiled function.

That completes the loop over all the tokens in the template. When the loop is done, all of the template has been processed. We have one last check to make: if `ops_stack` isn't empty, then we must be missing an end tag. Then we flush the buffered output to the function source:

```
        if ops_stack:
            self._syntax_error("Unmatched action tag", ops_stack[-1])

        flush_output()
```

We had created a section at the beginning of the function. Its role was to unpack template variables from the context into Python locals. Now that we've processed the entire template, we know the names of all the variables, so we can write the lines in this prologue.

We have to do a little work to know what names we need to define. Looking at our sample template:

```
<p>Welcome, {{user_name}}!</p>
<p>Products:</p>
<ul>
{% for product in product_list %}
    <li>{{ product.name }}:
        {{ product.price|format_price }}</li>
{% endfor %}
</ul>
```

There are two variables used here, `user_name` and `product`. The `all_vars` set will have both of those names, because both are used in `{{...}}` expressions. But only `user_name` needs to be extracted from the context in the prologue, because `product` is defined by the loop.

All the variables used in the template are in the set `all_vars`, and all the variables defined in the template are in `loop_vars`. All of the names in `loop_vars` have already been defined in the code because they are used in loops. So we need to unpack any name in `all_vars` that isn't in `loop_vars`:

```
        for var_name in self.all_vars - self.loop_vars:
            vars_code.add_line("c_%s = context[%r]" % (var_name, var_name))
```

Each name becomes a line in the function's prologue, unpacking the context variable into a suitably named local variable.

We're almost done compiling the template into a Python function. Our function has been appending strings to `result`, so the last line of the function is simply to join them all together and return them:

```
        code.add_line("return ''.join(result)")
        code.dedent()
```

Now that we've finished writing the source for our compiled Python function, we need to get the function itself from our CodeBuilder object. The `get_globals` method executes the Python code we've been assembling. Remember that our code is a function definition (starting with `def render_function(..):`), so executing the code will define `render_function`, but not execute the body of `render_function`.

The result of `get_globals` is the dictionary of values defined in the code. We grab the `render_function` value from it, and save it as an attribute in our Templite object:

```
        self._render_function = code.get_globals()['render_function']
```

Now `self._render_function` is a callable Python function. We'll use it later, during the rendering phase.

Compiling Expressions

We haven't yet seen a significant piece of the compiling process: the `_expr_code` method that compiles a template expression into a Python expression. Our template expressions can be as simple as a single name:

```
{{user_name}}
```

or can be a complex sequence of attribute accesses and filters:

```
{{user.name.localized|upper|escape}}
```

Our `_expr_code` method will handle all of these possibilities. As with expressions in any language, ours are built recursively: big expressions are composed of smaller expressions. A full expression is pipe-separated, where the first piece is dot-separated, and so on. So our function naturally takes a recursive form:

```
def _expr_code(self, expr):
    """Generate a Python expression for `expr`."""
```

The first case to consider is that our expression has pipes in it. If it does, then we split it into a list of pipe-pieces. The first pipe-piece is passed recursively to `_expr_code` to convert it into a Python expression.

```
    if "|" in expr:
        pipes = expr.split("|")
        code = self._expr_code(pipes[0])
        for func in pipes[1:]:
            self._variable(func, self.all_vars)
            code = "c_%s(%s)" % (func, code)
```

Each of the remaining pipe pieces is the name of a function. The value is passed through the function to produce the final value. Each function name is a variable that gets added to `all_vars` so that we can extract it properly in the prologue.

If there were no pipes, there might be dots. If so, split on the dots. The first part is passed recursively to `_expr_code` to turn it into a Python expression, then each dot name is handled in turn:

```
    elif "." in expr:
        dots = expr.split(".")
        code = self._expr_code(dots[0])
        args = ", ".join(repr(d) for d in dots[1:])
        code = "do_dots(%s, %s)" % (code, args)
```

To understand how dots get compiled, remember that `x.y` in the template could mean either `x['y']` or `x.y` in Python, depending on which works; if the result is callable, it's called. This uncertainty means that we have to try those possibilities at run time, not compile time. So we compile `x.y.z` into a function call, `do_dots(x, 'y', 'z')`. The dot function will try the various access methods and return the value that succeeded.

The `do_dots` function is passed into our compiled Python function at run time. We'll see its implementation in just a bit.

The last clause in the `_expr_code` function handles the case that there was no pipe or dot in the input expression. In that case, it's just a name. We record it in `all_vars`, and access the variable using its prefixed Python name:

```
    else:
        self._variable(expr, self.all_vars)
        code = "c_%s" % expr
    return code
```

Helper Functions

During compilation, we used a few helper functions. The `_syntax_error` method simply puts together a nice error message and raises the exception:

```
def _syntax_error(self, msg, thing):
    """Raise a syntax error using `msg`, and showing `thing`."""
    raise TempliteSyntaxError("%s: %r" % (msg, thing))
```

The `_variable` method helps us with validating variable names and adding them to the sets of names we collected during compilation. We use a regex to check that the name is a valid Python identifier, then add the name to the set:

```
def _variable(self, name, vars_set):
    """Track that `name` is used as a variable.

    Adds the name to `vars_set`, a set of variable names.

    Raises an syntax error if `name` is not a valid name.

    """
    if not re.match(r"[_a-zA-Z][_a-zA-Z0-9]*$", name):
        self._syntax_error("Not a valid name", name)
    vars_set.add(name)
```

With that, the compilation code is done!

Rendering

All that's left is to write the rendering code. Since we've compiled our template to a Python function, the rendering code doesn't have much to do. It has to get the data context ready, and then call the compiled Python code:

```
def render(self, context=None):
    """Render this template by applying it to `context`.

    `context` is a dictionary of values to use in this rendering.

    """
    # Make the complete context we'll use.
    render_context = dict(self.context)
    if context:
        render_context.update(context)
    return self._render_function(render_context, self._do_dots)
```

Remember that when we constructed the `Templite` object, we started with a data context. Here we copy it, and merge in whatever data has been passed in for this rendering. The copying is so that successive rendering calls won't see each others' data, and the merging is so that we have a single dictionary to use for data lookups. This is how we build one unified data context from the contexts provided when the template was constructed, with the data provided now at render time.

Notice that the data passed to `render` could overwrite data passed to the Templite constructor. That tends not to happen, because the context passed to the constructor has global-ish things like filter definitions and constants, and the context passed to `render` has specific data for that one rendering.

Then we simply call our compiled `render_function`. The first argument is the complete data context, and the second argument is the function that will implement the dot semantics. We use the same implementation every time: our own `_do_dots` method.

```
def _do_dots(self, value, *dots):
    """Evaluate dotted expressions at runtime."""
    for dot in dots:
        try:
            value = getattr(value, dot)
        except AttributeError:
            value = value[dot]
        if callable(value):
            value = value()
    return value
```

During compilation, a template expression like `x.y.z` gets turned into `do_dots(x, 'y', 'z')`. This function loops over the dot-names, and for each one tries it as an attribute, and if that fails, tries it as a key. This is what gives our single template syntax the flexibility to act as either `x.y` or `x['y']`. At each step, we also check if the new value is callable, and if it is, we call it. Once we're done with all the dot-names, the value in hand is the value we want.

Here we used Python argument unpacking again (`*dots`) so that `_do_dots` could take any number of dot names. This gives us a flexible function that will work for any dotted expression we encounter in the template.

Note that when calling `self._render_function`, we pass in a function to use for evaluating dot expressions, but we always pass in the same one. We could have made that code part of the compiled template, but it's the same eight lines for every template, and those eight lines are part of the definition of how templates work, not part of the details of a particular template. It feels cleaner to implement it like this than to have that code be part of the compiled template.

21.7 Testing

Provided with the template engine is a suite of tests that cover all of the behavior and edge cases. I'm actually a little bit over my 500-line limit: the template engine is 252 lines, and the tests are 275 lines. This is typical of well-tested code: you have more code in your tests than in your product.

21.8 What's Left Out

Full-featured template engines provide much more than we've implemented here. To keep this code small, we're leaving out interesting ideas like:

- Template inheritance and inclusion
- Custom tags
- Automatic escaping
- Arguments to filters
- Complex conditional logic like else and elif

- Loops with more than one loop variable
- Whitespace control

Even so, our simple template engine is useful. In fact, it is the template engine used in coverage.py to produce its HTML reports.

21.9 Summing up

In 252 lines, we've got a simple yet capable template engine. Real template engines have many more features, but this code lays out the basic ideas of the process: compile the template to a Python function, then execute the function to produce the text result.

A Simple Web Server

Greg Wilson

22.1 Introduction

The web has changed society in countless ways over the last two decades, but its core has changed very little. Most systems still follow the rules that Tim Berners-Lee laid out a quarter of a century ago. In particular, most web servers still handle the same kinds of messages they did then, in the same way.

This chapter will explore how they do that. At the same time, it will explore how developers can create software systems that don't need to be rewritten in order to add new features.

22.2 Background

Pretty much every program on the web runs on a family of communication standards called Internet Protocol (IP). The member of that family which concerns us is the Transmission Control Protocol (TCP/IP), which makes communication between computers look like reading and writing files.

Programs using IP communicate through sockets. Each socket is one end of a point-to-point communication channel, just like a phone is one end of a phone call. A socket consists of an IP address that identifies a particular machine and a port number on that machine. The IP address consists of four 8-bit numbers, such as `174.136.14.108`; the Domain Name System (DNS) matches these numbers to symbolic names like `aosabook.org` that are easier for human beings to remember.

A port number is a number in the range 0-65535 that uniquely identifies the socket on the host machine. (If an IP address is like a company's phone number, then a port number is like an extension.) Ports 0-1023 are reserved for the operating system's use; anyone else can use the remaining ports.

The Hypertext Transfer Protocol (HTTP) describes one way that programs can exchange data over IP. HTTP is deliberately simple: the client sends a request specifying what it wants over a socket connection, and the server sends some data in response (Figure 22.1.) The data may be copied from a file on disk, generated dynamically by a program, or some mix of the two.

The most important thing about an HTTP request is that it's just text: any program that wants to can create one or parse one. In order to be understood, though, that text must have the parts shown in Figure 22.2.

The HTTP method is almost always either "GET" (to fetch information) or "POST" (to submit form data or upload files). The URL specifies what the client wants; it is often a path to a file on disk, such as `/research/experiments.html`, but (and this is the crucial part) it's completely up to

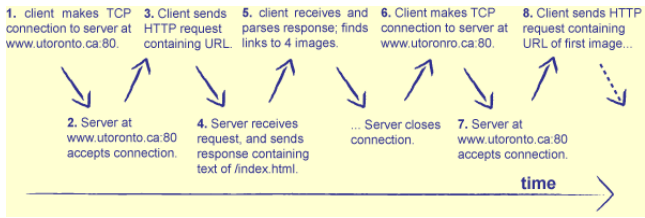

Figure 22.1: The HTTP Cycle

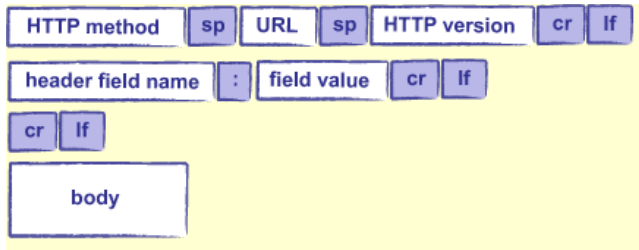

Figure 22.2: An HTTP Request

the server to decide what to do with it. The HTTP version is usually "HTTP/1.0" or "HTTP/1.1"; the differences between the two don't matter to us.

HTTP headers are key/value pairs like the three shown below:

```
Accept: text/html
Accept-Language: en, fr
If-Modified-Since: 16-May-2005
```

Unlike the keys in hash tables, keys may appear any number of times in HTTP headers. This allows a request to do things like specify that it's willing to accept several types of content.

Finally, the body of the request is any extra data associated with the request. This is used when submitting data via web forms, when uploading files, and so on. There must be a blank line between the last header and the start of the body to signal the end of the headers.

One header, called `Content-Length`, tells the server how many bytes to expect to read in the body of the request.

HTTP responses are formatted like HTTP requests (Figure 22.3):

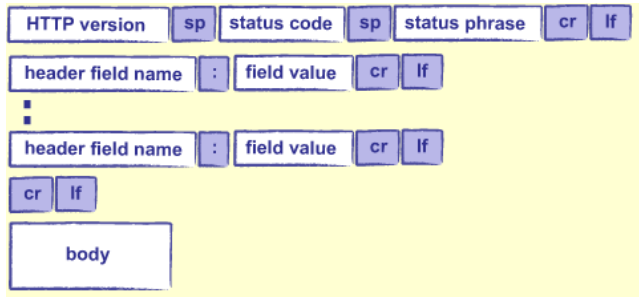

Figure 22.3: An HTTP Response

The version, headers, and body have the same form and meaning. The status code is a number indicating what happened when the request was processed: 200 means "everything worked", 404

means “not found”, and other codes have other meanings. The status phrase repeats that information in a human-readable phrase like “OK” or “not found”.

For the purposes of this chapter there are only two other things we need to know about HTTP.

The first is that it is *stateless*: each request is handled on its own, and the server doesn't remember anything between one request and the next. If an application wants to keep track of something like a user's identity, it must do so itself.

The usual way to do this is with a cookie, which is a short character string that the server sends to the client, and the client later returns to the server. When a user performs some function that requires state to be saved across several requests, the server creates a new cookie, stores it in a database, and sends it to her browser. Each time her browser sends the cookie back, the server uses it to look up information about what the user is doing.

The second thing we need to know about HTTP is that a URL can be supplemented with parameters to provide even more information. For example, if we're using a search engine, we have to specify what our search terms are. We could add these to the path in the URL, but what we should do is add parameters to the URL. We do this by adding ‘?’ to the URL followed by ‘key=value’ pairs separated by ‘&’. For example, the URL `http://www.google.ca?q=Python` asks Google to search for pages related to Python: the key is the letter ‘q’, and the value is ‘Python’. The longer query `http://www.google.ca/search?q=Python&client=Firefox` tells Google that we're using Firefox, and so on. We can pass whatever parameters we want, but again, it's up to the application running on the web site to decide which ones to pay attention to, and how to interpret them.

Of course, if ‘?’ and ‘&’ are special characters, there must be a way to escape them, just as there must be a way to put a double quote character inside a character string delimited by double quotes. The URL encoding standard represents special characters using ‘%’ followed by a 2-digit code, and replaces spaces with the ‘+’ character. Thus, to search Google for “grade = A+” (with the spaces), we would use the URL `http://www.google.ca/search?q=grade+%3D+A%2B`.

Opening sockets, constructing HTTP requests, and parsing responses is tedious, so most people use libraries to do most of the work. Python comes with such a library called `urllib2` (because it's a replacement for an earlier library called `urllib`), but it exposes a lot of plumbing that most people never want to care about. The Requests[1] library is an easier-to-use alternative to `urllib2`. Here's an example that uses it to download a page from the AOSA book site:

```
import requests
response = requests.get('http://aosabook.org/en/500L/web-server/testpage.html')
print 'status code:', response.status_code
print 'content length:', response.headers['content-length']
print response.text
```

```
status code: 200
content length: 61
<html>
  <body>
    <p>Test page.</p>
  </body>
</html>
```

`request.get` sends an HTTP GET request to a server and returns an object containing the response. That object's `status_code` member is the response's status code; its `content_length`

[1] `https://pypi.python.org/pypi/requests`

member is the number of bytes in the response data, and `text` is the actual data (in this case, an HTML page).

22.3 Hello, Web

We're now ready to write our first simple web server. The basic idea is simple:

1. Wait for someone to connect to our server and send an HTTP request;
2. parse that request;
3. figure out what it's asking for;
4. fetch that data (or generate it dynamically);
5. format the data as HTML; and
6. send it back.

Steps 1, 2, and 6 are the same from one application to another, so the Python standard library has a module called `BaseHTTPServer` that does those for us. We just have to take care of steps 3-5, which we do in the little program below:

```
import BaseHTTPServer

class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    '''Handle HTTP requests by returning a fixed 'page'.'''

    # Page to send back.
    Page = '''\
<html>
<body>
<p>Hello, web!</p>
</body>
</html>
'''

    # Handle a GET request.
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-Type", "text/html")
        self.send_header("Content-Length", str(len(self.Page)))
        self.end_headers()
        self.wfile.write(self.Page)

#----------------------------------------------------------------------

if __name__ == '__main__':
    serverAddress = ('', 8080)
    server = BaseHTTPServer.HTTPServer(serverAddress, RequestHandler)
    server.serve_forever()
```

The library's `BaseHTTPRequestHandler` class takes care of parsing the incoming HTTP request and deciding what method it contains. If the method is GET, the class calls a method named `do_GET`. Our class `RequestHandler` overrides this method to dynamically generate a simple page: the text is stored in the class-level variable `Page`, which we send back to the client after sending a 200 response

code, a `Content-Type` header telling the client to interpret our data as HTML, and the page's length. (The `end_headers` method call inserts the blank line that separates our headers from the page itself.)

But `RequestHandler` isn't the whole story: we still need the last three lines to actually start a server running. The first of these lines defines the server's address as a tuple: the empty string means "run on the current machine", and 8080 is the port. We then create an instance of `BaseHTTPServer.HTTPServer` with that address and the name of our request handler class as parameters, then ask it to run forever (which in practice means until we kill it with Control-C).

If we run this program from the command line, it doesn't display anything:

```
$ python server.py
```

If we then go to `http://localhost:8080` with our browser, though, we get this in our browser:

```
Hello, web!
```

and this in our shell:

```
127.0.0.1 - - [24/Feb/2014 10:26:28] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [24/Feb/2014 10:26:28] "GET /favicon.ico HTTP/1.1" 200 -
```

The first line is straightforward: since we didn't ask for a particular file, our browser has asked for '/' (the root directory of whatever the server is serving). The second line appears because our browser automatically sends a second request for an image file called `/favicon.ico`, which it will display as an icon in the address bar if it exists.

22.4 Displaying Values

Let's modify our web server to display some of the values included in the HTTP request. (We'll do this pretty frequently when debugging, so we might as well get some practice.) To keep our code clean, we'll separate creating the page from sending it:

```
class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):

    # ...page template...

    def do_GET(self):
        page = self.create_page()
        self.send_page(page)

    def create_page(self):
        # ...fill in...

    def send_page(self, page):
        # ...fill in...
```

`send_page` is pretty much what we had before:

```
    def send_page(self, page):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.send_header("Content-Length", str(len(page)))
        self.end_headers()
        self.wfile.write(page)
```

The template for the page we want to display is just a string containing an HTML table with some formatting placeholders:

```
    Page = '''\
<html>
<body>
<table>
<tr>  <td>Header</td>         <td>Value</td>          </tr>
<tr>  <td>Date and time</td>  <td>{date_time}</td>    </tr>
<tr>  <td>Client host</td>    <td>{client_host}</td>  </tr>
<tr>  <td>Client port</td>    <td>{client_port}s</td> </tr>
<tr>  <td>Command</td>        <td>{command}</td>      </tr>
<tr>  <td>Path</td>           <td>{path}</td>         </tr>
</table>
</body>
</html>
'''
```

and the method that fills this in is:

```
    def create_page(self):
        values = {
            'date_time'   : self.date_time_string(),
            'client_host' : self.client_address[0],
            'client_port' : self.client_address[1],
            'command'     : self.command,
            'path'        : self.path
        }
        page = self.Page.format(**values)
        return page
```

The main body of the program is unchanged: as before, it creates an instance of the `HTTPServer` class with an address and this request handler as parameters, then serves requests forever. If we run it and send a request from a browser for `http://localhost:8080/something.html`, we get:

```
Date and time  Mon, 24 Feb 2014 17:17:12 GMT
Client host    127.0.0.1
Client port    54548
Command        GET
Path           /something.html
```

Notice that we do *not* get a 404 error, even though the page `something.html` doesn't exist as a file on disk. That's because a web server is just a program, and can do whatever it wants when it gets a request: send back the file named in the previous request, serve up a Wikipedia page chosen at random, or whatever else we program it to.

22.5 Serving Static Pages

The obvious next step is to start serving pages from the disk instead of generating them on the fly. We'll start by rewriting `do_GET`:

```
def do_GET(self):
    try:

        # Figure out what exactly is being requested.
        full_path = os.getcwd() + self.path

        # It doesn't exist...
        if not os.path.exists(full_path):
            raise ServerException("'{0}' not found".format(self.path))

        # ...it's a file...
        elif os.path.isfile(full_path):
            self.handle_file(full_path)

        # ...it's something we don't handle.
        else:
            raise ServerException("Unknown object '{0}'".format(self.path))

    # Handle errors.
    except Exception as msg:
        self.handle_error(msg)
```

This method assumes that it's allowed to serve any files in or below the directory that the web server is running in (which it gets using `os.getcwd`). It combines this with the path provided in the URL (which the library automatically puts in `self.path`, and which always starts with a leading '/') to get the path to the file the user wants.

If that doesn't exist, or if it isn't a file, the method reports an error by raising and catching an exception. If the path matches a file, on the other hand, it calls a helper method named `handle_file` to read and return the contents. This method just reads the file and uses our existing `send_content` to send it back to the client:

```
def handle_file(self, full_path):
    try:
        with open(full_path, 'rb') as reader:
            content = reader.read()
        self.send_content(content)
    except IOError as msg:
        msg = "'{0}' cannot be read: {1}".format(self.path, msg)
        self.handle_error(msg)
```

Note that we open the file in binary mode—the 'b' in 'rb'—so that Python won't try to "help" us by altering byte sequences that look like a Windows line ending. Note also that reading the whole file into memory when serving it is a bad idea in real life, where the file might be several gigabytes of video data. Handling that situation is outside the scope of this chapter.

To finish off this class, we need to write the error handling method and the template for the error reporting page:

```
Error_Page = """\
    <html>
    <body>
    <h1>Error accessing {path}</h1>
```

```
        <p>{msg}</p>
        </body>
        </html>
        """

    def handle_error(self, msg):
        content = self.Error_Page.format(path=self.path, msg=msg)
        self.send_content(content)
```

This program works, but only if we don't look too closely. The problem is that it always returns a status code of 200, even when the page being requested doesn't exist. Yes, the page sent back in that case contains an error message, but since our browser can't read English, it doesn't know that the request actually failed. In order to make that clear, we need to modify `handle_error` and `send_content` as follows:

```
    # Handle unknown objects.
    def handle_error(self, msg):
        content = self.Error_Page.format(path=self.path, msg=msg)
        self.send_content(content, 404)

    # Send actual content.
    def send_content(self, content, status=200):
        self.send_response(status)
        self.send_header("Content-type", "text/html")
        self.send_header("Content-Length", str(len(content)))
        self.end_headers()
        self.wfile.write(content)
```

Note that we don't raise `ServerException` when a file can't be found, but generate an error page instead. A `ServerException` is meant to signal an internal error in the server code, i.e., something that *we* got wrong. The error page created by `handle_error`, on the other hand, appears when the *user* got something wrong, i.e., sent us the URL of a file that doesn't exist. [2]

22.6 Listing Directories

As our next step, we could teach the web server to display a listing of a directory's contents when the path in the URL is a directory rather than a file. We could even go one step further and have it look in that directory for an `index.html` file to display, and only show a listing of the directory's contents if that file is not present.

But building these rules into `do_GET` would be a mistake, since the resulting method would be a long tangle of `if` statements controlling special behaviors. The right solution is to step back and solve the general problem, which is figuring out what to do with a URL. Here's a rewrite of the `do_GET` method:

```
    def do_GET(self):
        try:
```

[2] We're going to use `handle_error` several times throughout this chapter, including several cases where the status code `404` isn't appropriate. As you read on, try to think of how you would extend this program so that the status response code can be supplied easily in each case.

```
            # Figure out what exactly is being requested.
            self.full_path = os.getcwd() + self.path

            # Figure out how to handle it.
            for case in self.Cases:
                handler = case()
                if handler.test(self):
                    handler.act(self)
                    break

        # Handle errors.
        except Exception as msg:
            self.handle_error(msg)
```

The first step is the same: figure out the full path to the thing being requested. After that, though, the code looks quite different. Instead of a bunch of inline tests, this version loops over a set of cases stored in a list. Each case is an object with two methods: `test`, which tells us whether it's able to handle the request, and `act`, which actually takes some action. As soon as we find the right case, we let it handle the request and break out of the loop.

These three case classes reproduce the behavior of our previous server:

```
class case_no_file(object):
    '''File or directory does not exist.'''

    def test(self, handler):
        return not os.path.exists(handler.full_path)

    def act(self, handler):
        raise ServerException("'{0}' not found".format(handler.path))


class case_existing_file(object):
    '''File exists.'''

    def test(self, handler):
        return os.path.isfile(handler.full_path)

    def act(self, handler):
        handler.handle_file(handler.full_path)


class case_always_fail(object):
    '''Base case if nothing else worked.'''

    def test(self, handler):
        return True

    def act(self, handler):
        raise ServerException("Unknown object '{0}'".format(handler.path))
```

and here's how we construct the list of case handlers at the top of the `RequestHandler` class:

```
class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    '''
    If the requested path maps to a file, that file is served.
    If anything goes wrong, an error page is constructed.
    '''

    Cases = [case_no_file(),
             case_existing_file(),
             case_always_fail()]

    ...everything else as before...
```

Now, on the surface this has made our server more complicated, not less: the file has grown from 74 lines to 99, and there's an extra level of indirection without any new functionality. The benefit comes when we go back to the task that started this chapter and try to teach our server to serve up the `index.html` page for a directory if there is one, and a listing of the directory if there isn't. The handler for the former is:

```
class case_directory_index_file(object):
    '''Serve index.html page for a directory.'''

    def index_path(self, handler):
        return os.path.join(handler.full_path, 'index.html')

    def test(self, handler):
        return os.path.isdir(handler.full_path) and \
               os.path.isfile(self.index_path(handler))

    def act(self, handler):
        handler.handle_file(self.index_path(handler))
```

Here, the helper method `index_path` constructs the path to the `index.html` file; putting it in the case handler prevents clutter in the main `RequestHandler`. `test` checks whether the path is a directory containing an `index.html` page, and `act` asks the main request handler to serve that page.

The only change needed to `RequestHandler` is to add a `case_directory_index_file` object to our `Cases` list:

```
    Cases = [case_no_file(),
             case_existing_file(),
             case_directory_index_file(),
             case_always_fail()]
```

What about directories that don't contain `index.html` pages? The test is the same as the one above with a `not` strategically inserted, but what about the `act` method? What should it do?

```
class case_directory_no_index_file(object):
    '''Serve listing for a directory without an index.html page.'''

    def index_path(self, handler):
        return os.path.join(handler.full_path, 'index.html')
```

```
    def test(self, handler):
        return os.path.isdir(handler.full_path) and \
               not os.path.isfile(self.index_path(handler))

    def act(self, handler):
        ???
```

It seems we've backed ourselves into a corner. Logically, the `act` method should create and return the directory listing, but our existing code doesn't allow for that: `RequestHandler.do_GET` calls `act`, but doesn't expect or handle a return value from it. For now, let's add a method to `RequestHandler` to generate a directory listing, and call that from the case handler's `act`:

```
class case_directory_no_index_file(object):
    '''Serve listing for a directory without an index.html page.'''

    # ...index_path and test as above...

    def act(self, handler):
        handler.list_dir(handler.full_path)


class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):

    # ...all the other code...

    # How to display a directory listing.
    Listing_Page = '''\
        <html>
        <body>
        <ul>
        {0}
        </ul>
        </body>
        </html>
        '''

    def list_dir(self, full_path):
        try:
            entries = os.listdir(full_path)
            bullets = ['<li>{0}</li>'.format(e)
                for e in entries if not e.startswith('.')]
            page = self.Listing_Page.format('\n'.join(bullets))
            self.send_content(page)
        except OSError as msg:
            msg = "'{0}' cannot be listed: {1}".format(self.path, msg)
            self.handle_error(msg)
```

22.7 The CGI Protocol

Of course, most people won't want to edit the source of their web server in order to add new functionality. To save them from having to do so, servers have always supported a mechanism called

the Common Gateway Interface (CGI), which provides a standard way for a web server to run an external program in order to satisfy a request.

For example, suppose we want the server to be able to display the local time in an HTML page. We can do this in a standalone program with just a few lines of code:

```
from datetime import datetime
print '''\
<html>
<body>
<p>Generated {0}</p>
</body>
</html>'''.format(datetime.now())
```

In order to get the web server to run this program for us, we add this case handler:

```
class case_cgi_file(object):
    '''Something runnable.'''

    def test(self, handler):
        return os.path.isfile(handler.full_path) and \
               handler.full_path.endswith('.py')

    def act(self, handler):
        handler.run_cgi(handler.full_path)
```

The test is simple: does the file path end with `.py`? If so, `RequestHandler` runs this program.

```
    def run_cgi(self, full_path):
        cmd = "python " + full_path
        child_stdin, child_stdout = os.popen2(cmd)
        child_stdin.close()
        data = child_stdout.read()
        child_stdout.close()
        self.send_content(data)
```

This is horribly insecure: if someone knows the path to a Python file on our server, we're just letting them run it without worrying about what data it has access to, whether it might contain an infinite loop, or anything else.[3]

Sweeping that aside, the core idea is simple:

1. Run the program in a subprocess.
2. Capture whatever that subprocess sends to standard output.
3. Send that back to the client that made the request.

The full CGI protocol is much richer than this—in particular, it allows for parameters in the URL, which the server passes into the program being run—but those details don't affect the overall architecture of the system...

...which is once again becoming rather tangled. `RequestHandler` initially had one method, `handle_file`, for dealing with content. We have now added two special cases in the form of

[3] Our code also uses the `popen2` library function, which has been deprecated in favor of the `subprocess` module. However, `popen2` was the less distracting tool to use in this example.

`list_dir` and `run_cgi`. These three methods don't really belong where they are, since they're primarily used by others.

The fix is straightforward: create a parent class for all our case handlers, and move other methods to that class if (and only if) they are shared by two or more handlers. When we're done, the `RequestHandler` class looks like this:

```
class RequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):

    Cases = [case_no_file(),
             case_cgi_file(),
             case_existing_file(),
             case_directory_index_file(),
             case_directory_no_index_file(),
             case_always_fail()]

    # How to display an error.
    Error_Page = """\
        <html>
        <body>
        <h1>Error accessing {path}</h1>
        <p>{msg}</p>
        </body>
        </html>
        """

    # Classify and handle request.
    def do_GET(self):
        try:

            # Figure out what exactly is being requested.
            self.full_path = os.getcwd() + self.path

            # Figure out how to handle it.
            for case in self.Cases:
                if case.test(self):
                    case.act(self)
                    break

        # Handle errors.
        except Exception as msg:
            self.handle_error(msg)

    # Handle unknown objects.
    def handle_error(self, msg):
        content = self.Error_Page.format(path=self.path, msg=msg)
        self.send_content(content, 404)

    # Send actual content.
    def send_content(self, content, status=200):
        self.send_response(status)
        self.send_header("Content-type", "text/html")
```

```
        self.send_header("Content-Length", str(len(content)))
        self.end_headers()
        self.wfile.write(content)
```

while the parent class for our case handlers is:

```
class base_case(object):
    '''Parent for case handlers.'''

    def handle_file(self, handler, full_path):
        try:
            with open(full_path, 'rb') as reader:
                content = reader.read()
            handler.send_content(content)
        except IOError as msg:
            msg = "'{0}' cannot be read: {1}".format(full_path, msg)
            handler.handle_error(msg)

    def index_path(self, handler):
        return os.path.join(handler.full_path, 'index.html')

    def test(self, handler):
        assert False, 'Not implemented.'

    def act(self, handler):
        assert False, 'Not implemented.'
```

and the handler for an existing file (just to pick an example at random) is:

```
class case_existing_file(base_case):
    '''File exists.'''

    def test(self, handler):
        return os.path.isfile(handler.full_path)

    def act(self, handler):
        self.handle_file(handler, handler.full_path)
```

22.8 Discussion

The differences between our original code and the refactored version reflect two important ideas. The first is to think of a class as a collection of related services. `RequestHandler` and `base_case` don't make decisions or take actions; they provide tools that other classes can use to do those things.

The second is extensibility: people can add new functionality to our web server either by writing an external CGI program, or by adding a case handler class. The latter does require a one-line change to `RequestHandler` (to insert the case handler in the `Cases` list), but we could get rid of that by having the web server read a configuration file and load handler classes from that. In both cases, they can ignore most lower-level details, just as the authors of the `BaseHTTPRequestHandler` class have allowed us to ignore the details of handling socket connections and parsing HTTP requests.

These ideas are generally useful; see if you can find ways to use them in your own projects.

Colophon

The cover font is Museo from the exljibris foundry, by Jos Buivenga. The text font is TEXGyre Termes and the heading font is TEXGyre Heros, both by Bogusław Jackowski and Janusz M. Nowacki. The code font is Inconsolata by Raph Levien.

The front cover photo is composed of twenty-three separate focus-stacked images of watch gear assemblies. The picture was taken by Kellar Wilson. (`http://kellarwilson.smugmug.com/`)

This book was built with open source software (with the exception of the cover). Programs like LATEX, Pandoc, and Python were especially helpful.